

SAS/OR® 14.1 User's Guide
Mathematical Programming



The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2015. *SAS/OR® 14.1 User's Guide: Mathematical Programming*. Cary, NC: SAS Institute Inc.

SAS/OR® 14.1 User's Guide: Mathematical Programming

Copyright © 2015, SAS Institute Inc., Cary, NC, USA

All Rights Reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

July 2015

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

Contents

Chapter 1. What's New in SAS/OR 14.1 1
Chapter 2. Using This Book 5
Chapter 3. Introduction to Optimization 11
Chapter 4. Shared Concepts and Topics 21
Chapter 5. The OPTMODEL Procedure 27
Chapter 6. The Constraint Programming Solver 193
Chapter 7. The Linear Programming Solver 255
Chapter 8. The Mixed Integer Linear Programming Solver 323
Chapter 9. The Network Solver 375
Chapter 10. The Nonlinear Programming Solver 487
Chapter 11. The Quadratic Programming Solver 545
Chapter 12. The OPTLP Procedure 571
Chapter 13. The OPTMILP Procedure 625
Chapter 14. The OPTQP Procedure 685
Chapter 15. The Decomposition Algorithm 715
Chapter 16. The OPTMILP Option Tuner 821
Chapter 17. The MPS-Format SAS Data Set 839

Subject Index **855**

Syntax Index **864**

Acknowledgments

Credits

Documentation

Writing	Ioannis Akrotirianakis, Hao Cheng, Philipp Christophel, Matthew Galati, Dmitry V. Golovashkin, Melanie Gratton, Joshua Griffin, Menal Guzelsoy, Jennie Hu, Tao Huang, Trevor Kearney, Zhifeng Li, Richard Liu, Leo Lopes, Amar Narisetty, Michelle Opp, Imre Pólik, Girish Ramachandra, Jack Rouse, Ben-Hao Wang, Kaihong Xu, Yan Xu, Wenwen Zhou
Editing	Anne Baxter, Ed Huddleston
Documentation Support	Tim Arnold, Natalie Baerlocher, Remya Chandran, Melanie Gratton, Richard Liu, Jianzhe Luo, Michelle Opp, Portia Parker, Girish Ramachandra, Daniel Underwood
Technical Review	Shahrzad Azizzadeh, Tonya Chapman, Donna Fulenwider, Bill Gjertsen, Tao Huang, Edward P. Hughes, John Jasperse, Rui Kang, Charles B. Kelly, Radhika Kulkarni, Yu-Min Lin, M. Muraleetharan, Bengt Pederson, Rob Pratt, Kaihong Xu, Lois Zhu

Software

In the following list, the names of the developers who currently support the procedure are listed first.

OPTMODEL procedure	Leo Lopes, Jack Rouse
CLP solver	Leo Lopes, Liping Cai, Gehan A. Corea, Lindsey Puryear, Jack Rouse, Tien-yi D. Shaw, Keqi Yan
LP simplex algorithms	Philipp Christophel, Matthew Galati, Imre Pólik, Ben-Hao Wang, Yan Xu
LP interior point algorithm	Hao Cheng

MILP solver	Philipp Christophel, Matthew Galati, Menal Guzelsoy, Amar Narisetty, Yan Xu
Network solver	Matthew Galati, Leo Lopes, Jack Rouse
NLP solver	Joshua Griffin, Tao Huang, Ben-Hao Wang, Wenwen Zhou
QP solver	Hao Cheng
OPTLP procedure	Hao Cheng, Matthew Galati, Imre Pólik, Ben-Hao Wang, Yan Xu
OPTMILP procedure	Philipp Christophel, Amar Narisetty, Yan Xu
OPTQP procedure	Hao Cheng, Wenwen Zhou
Decomposition algorithm	Matthew Galati
OPTMILP option tuner	Ben-Hao Wang
MPS-format SAS data set	Hao Cheng, Amar Narisetty
Linear algebra specialist	Alexander Andrianov

Support Groups

Software Testing	Shahrzad Azizzadeh, Wei Huang, Rui Kang, Yu-Min Lin, M. Muraleetharan, Sanjeewa Naranpanawe, Bengt Pederson, Aysegul Peker, Rob Pratt, Jennifer Sloan, Jonathan Stephenson, Kaihong Xu, Wei Zhang, Lois Zhu
Technical Support	Tonya Chapman

Acknowledgments

Many people have been instrumental in the development of SAS/OR software. The individuals acknowledged here have been especially helpful.

Richard Brockmeier	Union Electric Company
Ken Cruthers	Goodyear Tire & Rubber Company
Patricia Duffy	Auburn University

Richard A. Ehrhardt	University of North Carolina at Greensboro
Paul Hardy	Babcock & Wilcox
Don Henderson	ORI Consulting Group
Dave Jennings	Lockheed Martin
Vidyadhar G. Kulkarni	University of North Carolina at Chapel Hill
Wayne Maruska	Basin Electric Power Cooperative
Roger Perala	United Sugars Corporation
Bruce Reed	Auburn University
Charles Rissmiller	Lockheed Martin
David Rubin	University of North Carolina at Chapel Hill
John Stone	North Carolina State University
Keith R. Weiss	ICI Americas Inc.

The final responsibility for the SAS System lies with SAS Institute alone. We hope that you will always let us know your opinions about the SAS System and its documentation. It is through your participation that SAS software is continuously improved.

Chapter 1
What's New in SAS/OR 14.1

Contents

Overview 1
Mathematical Optimization Updates 2
Solver Performance Improvements 2
PROC OPTMODEL Improvements 2
Other Optimization and Related Improvements 3
Discrete-Event Simulation Updates 3

Overview

SAS/OR 14.1 adds a number of new optimization capabilities that shorten optimization time, increase diagnostic capabilities, and make the software easier to use. Highlights include the following:

- Several solvers improve their performance.
- The concurrent FOR loop (the COFOR loop) in PROC OPTMODEL can run in distributed mode. **NOTE:** Distributed mode requires SAS High-Performance Optimization.
- PROC OPTMODEL adds a profiler that tracks the amount of time spent in problem generation, presolve, and various stages of the solution process.
- PROC OPTNET enables parallel computing, provides faster graph data input, and adds enhancements to three of its algorithms.
- The quadratic and nonlinear solvers add irreducible infeasible set (IIS) diagnostics.
- The decomposition algorithm expands the range of constraint matrix structures that it can detect automatically.
- The CLP procedure adds more variable selection strategies.

SAS Simulation Studio 14.1, a component of SAS/OR 14.1 for Windows environments, adds features that improve the accuracy of your models and give you additional controls on model execution. Highlights include the following:

- controls on the order in which dynamically created data input and output ports on blocks execute during the run of a model

- centralized controls on the ranking of blocks in a model, which determines the order of execution for events that are scheduled for the same simulation clock time
- expanded and improved controls on the allocation of resource units among resource entities when there is a scheduled adjustment
- automatic launching of the SAS server on your local PC

Mathematical Optimization Updates

Solver Performance Improvements

Several optimization solvers have improved their performance over that of SAS/OR 13.2 as follows:

- The interior point linear programming (LP) solver is 30% faster.
- The branch-and-cut mixed integer linear programming (MILP) solver is 35% faster.
- When METHOD=AUTO (the default setting), the decomposition (DECOMP) algorithm is 130% faster in finding an optimal MILP solution and 240% faster in finding a solution with a 5% optimality gap.
- The active set algorithm in the nonlinear programming (NLP) solver is 4% faster for the entire test suite and 10% faster for the more difficult problems in the test suite.

PROC OPTMODEL Improvements

The OPTMODEL procedure makes three major changes:

- The COFOR statement (which specifies a concurrent FOR loop) can distribute processing of solver invocations in the loop across multiple computational nodes, not simply among multiple computational threads on the same node. You request processing across multiple nodes by specifying a value greater than 1 in the NODES= option in the PERFORMANCE statement. **NOTE:** Distributed mode requires SAS High-Performance Optimization.
- CLP solver invocation via the SOLVE WITH CLP statement is promoted to production status.
- The PROFILE statement is added. This statement invokes and configures the PROC OPTMODEL profiler to track and report the time spent processing declarations, the time spent executing statements, and the number of times statements are executed. The profiler can help you identify elements of problem generation, presolve, and solution that require large amounts of time; it can be a useful aid in error detection and performance improvement. The PROFILE statement includes two optional arguments, *mode* and *options*. The *mode* argument enables or disables the profiler, and can also request that accumulated profiler data be printed. The *options* argument controls how PROC OPTMODEL collects, retains, and displays profiler information.

Other Optimization and Related Improvements

PROC CLP adds the new conflict-directed variable selection strategies, VARSELECT=WDEG and VARSELECT=DOMWDEG; adds a new dynamic variable section strategy, VARSELECT=DOMDDEG; and promotes the PACK and LEXICO constraint classes to production status.

PROC OPTNET enables faster graph data input. The STANDARDIZED_LABELS option in the PROC OPTNET statement enables you to read graph data that contain only numeric node identifiers more quickly. More generally, PROC OPTNET supports parallel computing, including parallel graph data input. This is especially helpful in shortening the time that is needed to input large-scale graph data. You can use the PERFORMANCE statement and its NTHREADS= option to request multithreaded computing.

PROC OPTNET adds enhancements to three of the algorithms that it uses. The TSP (traveling salesman problem) algorithm can solve asymmetric problems, which are defined on directed graphs; the shortest path algorithm can accept negative link weights; and the default connected components algorithm for undirected graphs is the more efficient union-find algorithm.

PROC OPTNET produces ODS tables as output. You can use the DETAILS option in the PERFORMANCE statement to request that PROC OPTNET produce the Timing ODS table, which reports the amount of time that each step of the procedure uses.

The NLP solver and the quadratic programming (QP) solver each add the IIS= option, which directs the solver to identify an irreducible infeasible set among the linear constraints and decision variable bounds for a problem that is found to be infeasible. Identification of irreducible infeasible sets provides valuable guidance in restoring infeasible problems to feasibility.

The MILP solver now runs in parallel (threaded or distributed) mode by default for improved performance. **NOTE:** Distributed mode requires SAS High-Performance Optimization.

The DECOMP algorithm for the LP and MILP solvers adds the SET value for the METHOD= option. This value directs the DECOMP algorithm to find a set-partitioning or set-covering structure in the constraint matrix. If such constraints are detected, they serve as the linking constraints, and the remaining weakly connected components of the constraint matrix define blocks of constraints for use in the DECOMP algorithm.

Discrete-Event Simulation Updates

SAS Simulation Studio 14.1, which provides a graphical environment for building and working with discrete-event simulation models, makes four major improvements:

- Three blocks have added new controls on the order in which dynamically created data input or output ports (or both) execute during the run of a model. Correct ordering of port execution can be critical to ensuring that your model functions properly. In previous releases, only the Formula block provided this feature, and the dynamic ports on other blocks executed in the order in which they were created. In SAS Simulation Studio 14.1, the Modifier, Extractor, and Gate blocks also include buttons, labeled **Move Up** and **Move Down**, that enable you to increase or decrease, respectively, the priority of any dynamic port in the order of execution.

- Centralized controls on the ranking of blocks in a model enable you to determine the order of execution for events that are scheduled for the same simulation clock time. These controls are similar in nature to the controls for port execution order, but on a larger scale. The model-wide Block Ranking dialog box enables you to specify tie-breaking priority rules to determine which block's events execute first. You open this dialog box by right-clicking the name of a model and selecting **Block Ranking**. The Block Ranking dialog box displays in a hierarchical check-box tree all blocks that have an adjustable rank in the selected model. A horizontal bar represents the current rank for each such block. Selecting the check box for a block enables you to drag its bar to adjust its rank. A longer horizontal bar corresponds to a higher rank. All ranks correspond to specific numeric values (and can also be specified numerically), but the rank values are evaluated only relatively for practical purposes. For example, if the rank of block A is higher than the rank of block B, the precise values involved are irrelevant; if block A and block B have events scheduled for the same simulation clock time, then the block A event will execute first.

- Controls on the allocation of resource units among resource entities have been expanded and improved. The Resource Scheduler block dialog box adds the **Adjust Units By** field, enabling you to choose the policy that is used to increase or decrease the units of resource entities when there is a scheduled adjustment. You can choose from among the following allocation policies:

  - **FairIntegerBased:** Any adjustments to the units of a resource entity are made on an integer basis, and any fractional resource units are ignored. If resource units increase or decrease, then those units are distributed integrally and as evenly as possible among all potential resource entities. This is the new default allocation policy.
  - **IntegerBased:** For a scheduled decrease in resource units, the units of the first targeted resource entity are integrally decreased as much as possible. If necessary, other resource entities are targeted if the first entity does not have enough units to cover the scheduled decrease. For an increase in resource units, this policy is the same as the **FairIntegerBased** policy.
  - **FractionBased:** For a scheduled increase in resource units, the additional units are distributed evenly among all targeted resource entities, potentially resulting in fractional units for some resource entities. For a scheduled decrease in resource units, the behavior is the same as for the **IntegerBased** policy, except that fractional units are also included. This was the default allocation policy in releases prior to SAS Simulation Studio 14.1.

- The SAS server on the local PC on which SAS Simulation Studio is installed is automatically launched on an as-needed basis. In previous releases, you were required to launch the local SAS server before reading a SAS data set, writing simulated data to a SAS data set, or running a SAS program. This enhancement ensures that the local SAS server will be available whenever it is required. However, if you specify that any block in your model should use a remote SAS workspace server, you are still required to start that server by using the Remote SAS Workspace Server Login dialog box, which appears during model execution.

Chapter 2
Using This Book

Contents

Purpose 5
Organization 5
Typographical Conventions 7
Conventions for Examples 7
Accessing the SAS/OR Sample Library 8
Online Documentation 8
Additional Documentation for SAS/OR Software 8

Purpose

SAS/OR User's Guide: Mathematical Programming provides a complete reference for the mathematical programming procedures in SAS/OR software. This book serves as the primary documentation for the OPTLP, OPTMILP, OPTMODEL, and OPTQP procedures; the various solvers used by PROC OPTMODEL; and the MPS-format SAS data set specification.

This chapter describes the organization of this book and the conventions that are used in the text and example code. To gain full benefit from using this book, you should familiarize yourself with the information presented in this section and refer to it when needed. The section "Additional Documentation for SAS/OR Software" on page 8 refers to other documents that contain related information.

Organization

Chapter 3, "Introduction to Optimization," contains a brief overview of the mathematical programming procedures in SAS/OR software and provides an introduction to optimization and the use of the optimization tools in the SAS System. That chapter also describes the flow of data between the procedures and how the components of the SAS System fit together.

Chapter 4, "Shared Concepts and Topics," details syntax that is common to all the procedures in this book. The chapter also reviews other topics such as ODS output and parallel processing that are not specific to one procedure.

Chapter 5, "The OPTMODEL Procedure," describes the OPTMODEL procedure, and the five subsequent chapters describe various solvers (linear programming, mixed integer linear programming, nonlinear programming, quadratic programming, and network) that the OPTMODEL procedure uses. The next three

chapters describe the OPTLP, OPTMILP, and OPTQP procedures for solving linear programming, mixed integer linear programming, and quadratic programming problems, respectively. The next two chapters describe the decomposition algorithm for linear and mixed integer linear programming and the option tuner for the OPTMILP procedure. The final chapter is the specification of the MPS-format SAS data set.

Each procedure description is self-contained; you need to be familiar with only the basic features of the SAS System and with SAS terminology to use most procedures. The statements and syntax necessary to run each procedure are presented in a uniform format throughout this book.

The following list summarizes the types of information provided for each procedure:

Overview provides a general description of what the procedure does. It outlines major capabilities of the procedure and lists all input and output data sets that are used with it.

Getting Started illustrates simple uses of the procedure in a few short examples. It provides introductory hands-on information for the procedure.

Syntax constitutes the major reference section for the syntax of the procedure. First, the statement syntax is summarized. Next, a functional summary table lists all the statements and options in the procedure, classified by function. In addition, the online version includes a Dictionary of Options, which provides an alphabetical list of all options. Following these tables, the PROC statement is described, and then all other statements are described in alphabetical order.

Details describes the features of the procedure, including algorithmic details and computational methods. It also explains how the various options interact with each other. This section describes input and output data sets in greater detail, with definitions of the output variables, and explains the format of printed output, if any.

Examples consists of examples that are designed to illustrate the use of the procedure. Each example includes a description of the problem and lists the options that are highlighted by the example. The example shows the data and the SAS statements needed, and includes the output that is produced. You can duplicate the examples by copying the statements and data and running the SAS program. The SAS Sample Library contains the code that is used to run the examples shown in this book; consult your SAS Software representative for specific information about the Sample Library.

References lists references that are relevant to the chapter.

Typographical Conventions

This book uses various type styles, as explained by the following list:

roman	is the standard type style used for most text.
UPPERCASE ROMAN	is used for SAS statements, options, and other SAS language elements when they appear in the text. However, you can enter these elements in your own SAS code in lowercase, uppercase, or a mixture of the two. This style is also used for identifying arguments and values (in the syntax specifications) that are literals (for example, to denote valid keywords for a specific option).
UPPERCASE BOLD	is used in the "Syntax" section to identify SAS keywords, such as the names of procedures, statements, and options.
VariableName	is used for the names of SAS variables and data sets when they appear in the text.
oblique	is used to indicate an option variable for which you must supply a value (for example, DUPLICATE=*dup* indicates that you must supply a value for *dup*).
italic	is used for terms that are defined in the text, for emphasis, and for publication titles.
monospace	is used to show examples of SAS statements. In most cases, this book uses lowercase type for SAS code. You can enter your own SAS code in lowercase, uppercase, or a mixture of the two.

Conventions for Examples

Most of the output shown in this book is produced with the following SAS System options:

```
options linesize=80 pagesize=60 nonumber nodate;
```

Accessing the SAS/OR Sample Library

The SAS/OR Sample Library includes many examples that illustrate the use of SAS/OR software, including the examples used in this documentation. To access these sample programs from the SAS windowing environment, select **Help** from the main menu and then select **Getting Started with SAS Software**. On the **Contents** tab, expand the **Learning to Use SAS**, **Sample SAS Programs**, and **SAS/OR** items. Then click **Samples**.

Online Documentation

This documentation is available online with the SAS System. To access SAS/OR documentation from the SAS windowing environment, select **Help** from the main menu and then select **SAS Help and Documentation**. On the **Contents** tab, expand the **SAS Products** and **SAS/OR** items. Then expand the book you want to view. You can search the documentation by using the **Search** tab.

You can also access the documentation by going to `http://support.sas.com/documentation`.

Additional Documentation for SAS/OR Software

In addition to *SAS/OR User's Guide: Mathematical Programming,* you might find the following documents helpful when using SAS/OR software:

SAS/OR User's Guide: Bill of Material Processing
: provides documentation for the BOM procedure and all bill of material postprocessing SAS macros. The BOM procedure and SAS macros enable you to generate different reports and to perform several transactions to maintain and update bills of material.

SAS/OR User's Guide: Constraint Programming
: provides documentation for the constraint programming procedure in SAS/OR software. This book serves as the primary documentation for the CLP procedure.

SAS/OR User's Guide: Local Search Optimization
: provides documentation for the local search optimization procedures in SAS/OR software. This book serves as the primary documentation for the GA procedure, which uses genetic algorithms to solve optimization problems, and the OPTLSO procedure, which performs parallel hybrid derivative-free optimization.

SAS/OR User's Guide: Mathematical Programming Examples
: supplements the *SAS/OR User's Guide: Mathematical Programming* with additional examples that demonstrate best practices for building and solving linear programming, mixed integer linear programming, and quadratic programming problems. The problem statements are reproduced with permission from the book *Model Building in Mathematical Programming* by H. Paul Williams.

SAS/OR User's Guide: Mathematical Programming Legacy Procedures

provides documentation for the older mathematical programming procedures in SAS/OR software. This book serves as the primary documentation for the INTPOINT, LP, NETFLOW, and NLP procedures. Guidelines are also provided on migrating from these older procedures to the newer OPTMODEL family of procedures.

SAS/OR User's Guide: Network Optimization Algorithms

provides documentation for a set of algorithms that can be used to investigate the characteristics of networks and to solve network-oriented optimization problems. This book also documents PROC OPTNET, which invokes these algorithms and provides network-structured formats for input and output data.

SAS/OR User's Guide: Project Management

provides documentation for the project management procedures in SAS/OR software. This book serves as the primary documentation for the CPM, DTREE, GANTT, NETDRAW, and PM procedures, in addition to the PROJMAN Application, a graphical user interface for project management.

SAS Simulation Studio: User's Guide

provides documentation about using SAS Simulation Studio, a graphical application for creating and working with discrete-event simulation models. This book describes in detail how to build and run simulation models and how to interact with SAS software for analysis and with JMP software for experimental design and analysis.

Chapter 3
Introduction to Optimization

Contents

Overview **11**
Linear Programming Problems **13**
The OPTLP Procedure 13
The OPTMODEL Procedure 13
Mixed Integer Linear Problems **14**
The OPTMILP Procedure 14
The OPTMODEL Procedure 14
Quadratic Programming Problems **14**
The OPTQP Procedure 14
The OPTMODEL Procedure 15
Nonlinear Problems **15**
The OPTMODEL Procedure 15
Model Building with PROC OPTMODEL **15**
References **19**

Overview

Operations research tools are directed toward the solution of resource management and planning problems. Models in operations research are representations of the structure of a physical object or a conceptual or business process. Using the tools of operations research involves the following:

- defining a structural model of the system under investigation
- collecting the data for the model
- solving the model
- interpreting the results

SAS/OR software is a set of procedures for exploring models of distribution networks, production systems, resource allocation problems, and scheduling problems using the tools of operations research.

The following list suggests some of the application areas in which optimization-based decision support systems have been used. In practice, models often contain elements of several applications listed here.

- **Product-mix problems** find the mix of products that generates the largest return when several products compete for limited resources.
- **Blending problems** find the mix of ingredients to be used in a product so that it meets minimum standards at minimum cost.
- **Time-staged problems** are models whose structure repeats as a function of time. Production and inventory models are classic examples of time-staged problems. In each period, production plus inventory minus current demand equals inventory carried to the next period.
- **Scheduling problems** assign people to times, places, or tasks so as to optimize people's preferences or performance while satisfying the demands of the schedule.
- **Multiple objective problems** have multiple, possibly conflicting, objectives. Typically, the objectives are prioritized, and the problems are solved sequentially in a priority order.
- **Capital budgeting and project selection problems** ask for the project or set of projects that yield the greatest return.
- **Location problems** seek the set of locations that meets the distribution needs at minimum cost.
- **Cutting stock problems** find the partition of raw material that minimizes waste and fulfills demand.

The basic optimization problem is that of minimizing or maximizing an objective function subject to constraints imposed on the variables of that function. The objective function and constraints can be linear or nonlinear; the constraints can be bound constraints, equality or inequality constraints, or integer constraints. Traditionally, optimization problems are divided into various types depending on the sets of values that the variables are restricted to (real, integer, or binary, or a combination) and the nature of functional form of the constraints and objectives (linear, quadratic, or general nonlinear). An expression of an optimization problem in mathematical form is called a mathematical program.

When the complete description of a mathematical program is supplied to an appropriate algorithm (such as one of the solvers described in this book), the algorithm determines the optimal values for the decision variables so the objective is either maximized or minimized, the optimal values that are assigned to decision variables are on or between allowable bounds, and the constraints are obeyed. This process of solving mathematical programs is called mathematical programming, mathematical optimization, or just optimization.

When the constraints in an optimization problem are linear and the objective is either linear or quadratic, the optimization problem can be encapsulated in SAS data sets and then solved using the appropriate SAS/OR procedure: the OPTLP, OPTMILP, or OPTQP procedure.

Often optimization problems, and especially those with nonlinear elements, are formalized in an algebraic model that represents the problem. When formulated in its most abstract form, such an algebraic model is independent of problem data. A specific optimization problem instance (including the original problem) is then just an instantiation of the algebraic model with the specific data associated with that instance. An optimization modeling language (also called an algebraic modeling language) is a programming environment that has syntax, structures, and operations that enable you to express a mathematical program in a form that corresponds in a natural and transparent way to its algebraic model. The syntax, structures, and operations

also enable you to populate an algebraic model with a specific data instance and then solve the resulting optimization problem instance with an appropriate solver. The OPTMODEL procedure is such an algebraic modeling language in SAS/OR software and can be viewed as a single, unified environment to formulate and solve mathematical programming problems of many different types.

Whether mathematical programs are represented in SAS data sets or in an algebraic model in PROC OPTMODEL, they can be saved, easily changed, and solved again. The SAS/OR procedures also output SAS data sets that contain the solutions. These data sets can then be used to produce customized reports or as input to other SAS procedures. This structure enables you to use the tools of operations research and other SAS tools as building blocks to build decision support systems.

This chapter describes how to use SAS/OR software to solve a wide variety of optimization problems. It describes various types of optimization problems, indicates which SAS/OR procedures you can use, and shows how you provide data, run the procedure, and obtain optimal solutions. For additional examples that demonstrate the features of the OPTMODEL procedure, see *SAS/OR User's Guide: Mathematical Programming Examples*.

The next section broadly classifies the SAS/OR procedures based on the types of mathematical programming problems they can solve.

Linear Programming Problems

The OPTLP Procedure

The OPTLP procedure solves linear programming problems that are submitted in a SAS data set that uses a mathematical programming system (MPS) format.

The MPS file format is a format commonly used for describing linear programming (LP) and integer programming (IP) problems (Murtagh 1981; IBM 1988). MPS-format files are in text format and have specific conventions for the order in which the different pieces of the mathematical model are specified. The MPS-format SAS data set corresponds closely to the MPS file format and is used to describe linear programming problems for PROC OPTLP. For more details, see Chapter 17, "The MPS-Format SAS Data Set."

PROC OPTLP provides three solvers to solve general LPs: primal simplex, dual simplex, and interior point. The simplex solvers implement a two-phase simplex method, and the interior point solver implements a primal-dual predictor-corrector algorithm. For pure network LPs or LPs with significant network structure and additional linear side constraints, PROC OPTLP also provides a network simplex based solver. For more details about solving LPs with PROC OPTLP, see Chapter 12, "The OPTLP Procedure."

The OPTMODEL Procedure

The OPTMODEL procedure, a general purpose optimization modeling language, can also be used for concisely modeling linear programming problems. If an LP has special network structure, the structure is typically natural and evident in a well-formulated model of the problem in PROC OPTMODEL.

Within PROC OPTMODEL you can declare a model, pass it directly to various solvers, and review the solver result. You can also save an instance of a linear model in data set form for use by the OPTLP procedure. For more details, see Chapter 5, "The OPTMODEL Procedure."

Mixed Integer Linear Problems

The OPTMILP Procedure

The OPTMILP procedure solves general mixed integer linear programs (MILPs) —linear programs in which a subset of the decision variables are constrained to be integers. The OPTMILP procedure solves MILPs with an LP-based branch-and-bound algorithm augmented by advanced techniques such as cutting planes and primal heuristics. For more details about the OPTMILP procedure, see Chapter 13, "The OPTMILP Procedure."

The OPTMILP procedure requires a MILP to be specified by a SAS data set that adheres to the MPS format. See Chapter 17, "The MPS-Format SAS Data Set," for details about the MPS-format data set.

The OPTMODEL Procedure

The OPTMODEL procedure, a general purpose optimization modeling language, can also be used for concisely modeling mixed integer linear programming problems. In fact, except for the declaration of some subset of variables to be integer or binary, modeling these problems is quite analogous to modeling LPs. Within OPTMODEL you can declare a model, pass it directly to various solvers, and review the solver result. You can also save an instance of a mixed integer linear model in data set form for use by PROC OPTMILP. For more details, see Chapter 5, "The OPTMODEL Procedure."

Quadratic Programming Problems

The OPTQP Procedure

The OPTQP procedure solves quadratic programs—problems with a quadratic objective function and a collection of linear constraints, including general linear constraints along with lower or upper bounds (or both) on the decision variables.

You can specify the problem input data in one SAS data set that uses a quadratic programming system (QPS) format. For details about the QPS-format data specification, see Chapter 17, "The MPS-Format SAS Data Set." For more details about the OPTQP procedure, see Chapter 14, "The OPTQP Procedure."

The OPTMODEL Procedure

The OPTMODEL procedure, a general purpose optimization modeling language, can also be used for concisely modeling quadratic programming problems. Within OPTMODEL you can declare a model, pass it directly to various solvers, and review the solver result. You can also save an instance of a quadratic model in data set form for use by PROC OPTQP. For more details, see Chapter 5, "The OPTMODEL Procedure."

Nonlinear Problems

The OPTMODEL Procedure

The OPTMODEL procedure, a general purpose optimization modeling language, can also be used for concisely modeling nonlinear programming problems. Within OPTMODEL you can declare a nonlinear optimization model, pass it directly to various solvers, and review the solver result. For more details, see Chapter 5, "The OPTMODEL Procedure."

You can solve many different types of nonlinear programming problems with PROC OPTMODEL using its nonlinear solver functionality. For more details about the nonlinear programming solver, see Chapter 10, "The Nonlinear Programming Solver."

Model Building with PROC OPTMODEL

Model generation and maintenance are often difficult and expensive aspects of applying mathematical programming techniques. The richly expressive syntax and features of PROC OPTMODEL, in addition to the flexible data input and output capabilities, simplify this task considerably. Although PROC OPTMODEL offers almost unlimited latitude in how a particular optimization problem is formulated, the most effective use of OPTMODEL is achieved when the model is abstracted away from the data. This aspect makes PROC OPTMODEL somewhat unusual among SAS procedures and is important enough to illustrate with a simple example.

A small product-mix problem serves as a starting point for a discussion of two different ways of modeling with PROC OPTMODEL.

A candy manufacturer makes two products: chocolate and toffee. What combination of chocolate and toffee should be produced in a day in order to maximize the company's profit? Chocolate contributes \$0.25 per pound to profit, and toffee contributes \$0.75 per pound. The decision variables are *chocolate* and *toffee*.

Four processes are used to manufacture the candy:

1. Process 1 combines and cooks the basic ingredients for both chocolate and toffee.
2. Process 2 adds colors and flavors to the toffee, then cools and shapes the confection.

3. Process 3 chops and mixes nuts and raisins, adds them to the chocolate, and then cools and cuts the bars.

4. Process 4 is packaging: chocolate is placed in individual paper shells; toffee is wrapped in cellophane packages.

During the day, there are 7.5 hours (27,000 seconds) available for each process.

Firm time standards have been established for each process. For Process 1, mixing and cooking take 15 seconds for each pound of chocolate, and 40 seconds for each pound of toffee. Process 2 takes 56.25 seconds per pound of toffee. For Process 3, each pound of chocolate requires 18.75 seconds of processing. In packaging, a pound of chocolate can be wrapped in 12 seconds, whereas a pound of toffee requires 50 seconds. These data are summarized as follows:

	Process	Available Time (sec)	Required per Pound chocolate (sec)	Required per Pound toffee (sec)
1	Cooking	27,000	15	40
2	Color/Flavor	27,000		56.25
3	Condiments	27,000	18.75	
4	Packaging	27,000	12	50

The objective is to maximize the company's total profit, which is represented as

Maximize: $0.25(\textit{chocolate}) + 0.75(\textit{toffee})$

The production of the candy is limited by the time available for each process. The limits placed on production by Process 1 are expressed by the following inequality:

Process 1: $15(\textit{chocolate}) + 40(\textit{toffee}) \leq 27{,}000$

Process 1 can handle any combination of chocolate and toffee that satisfies this inequality.

The limits on production by other processes generate constraints described by the following inequalities:

Process 2: $56.25(\textit{toffee}) \leq 27{,}000$

Process 3: $18.75(\textit{chocolate}) \leq 27{,}000$

Process 4: $12(\textit{chocolate}) + 50(\textit{toffee}) \leq 27{,}000$

This linear program illustrates an example of a product mix problem. The mix of products that maximizes the objective without violating the constraints is the solution.

First, the following statements demonstrate a way of representing the optimization model in PROC OPTMODEL that is almost a verbatim translation of the mathematical model:

```
proc optmodel;
   /* declare variables */
   var choco >= 0, toffee >= 0;

   /* maximize objective function (profit) */
   maximize profit = 0.25*choco + 0.75*toffee;
```

```
   /* subject to constraints */
   con process1:    15*choco +    40*toffee <= 27000;
   con process2:               56.25*toffee <= 27000;
   con process3: 18.75*choco                <= 27000;
   con process4:    12*choco +    50*toffee <= 27000;

   /* solve LP using primal simplex solver */
   solve with lp / solver = primal_spx;

   /* display solution */
   print choco toffee;
quit;
```

The optimal objective value and the optimal solution are displayed in Figure 3.1:

Figure 3.1 Solution Summary

The OPTMODEL Procedure

Solution Summary	
Solver	LP
Algorithm	Primal Simplex
Objective Function	profit
Solution Status	Optimal
Objective Value	475
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	6
Presolve Time	0.00
Solution Time	0.00

choco	toffee
1000	300

You can observe from the preceding example that PROC OPTMODEL provides an easy and very direct way of modeling and solving mathematical programming models. Although this way of modeling, where the data are intertwined heavily with model elements, is correct, has significant practical limitations. The model is not easy to explain, it is hard to generalize, and clearly this approach does not scale to large problems of the same type. To overcome these issues, you need to separate the data from the essential algebraic structure of the model. Along those lines, you can make the reasonable assumption that you have the following two data sets (one for the products and one for processes that capture the parameters and data elements of this product mix problem):

```
data Products;
   length Name $10.;
   input Name $ Profit;
datalines;
Chocolate  0.25
```

```
Toffee      0.75
;

data Processes;
   length Name $15.;
   input Name $ Available_time Chocolate Toffee;
datalines;
Cooking             27000           15          40
Color/Flavor        27000            0          56.25
Condiments          27000          18.75         0
Packaging           27000           12          50
;
```

The following alternative model in PROC OPTMODEL can solve the same problem by taking these data sets as input:

```
proc optmodel;
   /* declare sets and data indexed by sets */
   set <string> Products;
   set <string> Processes;
   num Profit{Products};
   num AvailableTime{Processes};
   num RequiredTime{Products,Processes};
   /* declare the variable */
   var Amount{Products};
   /* maximize objective function (profit) */
   maximize TotalProfit = sum{p in Products} Profit[p]*Amount[p];
   /* subject to constraints */
   con Availability{r in Processes}:
      sum{p in Products} RequiredTime[p,r]*Amount[p] <= AvailableTime[r];
   /* abstract algebraic model that captures the structure of the */
   /*    optimization problem has been defined without referring  */
   /*    to a single data constant                                */
   /* populate model by reading in the specific data instance */
   read data Products into Products=[name] Profit;
   read data Processes into Processes=[name] AvailableTime=Available_time
      {p in Products} <RequiredTime[p,name]= col(p)>;
   /* solve LP using primal simplex solver */
   solve with lp / solver = primal_spx;
   /* display solution */
   print Amount;
quit;
```

The details of the syntax and elements of the PROC OPTMODEL language are discussed in Chapter 5, "The OPTMODEL Procedure." The key observation here is that the preceding version of the PROC OPTMODEL statements capture the essence of the optimization model concisely, but completely, and the model can be explained, modified, and maintained easily. It also achieves total separation of the data from the model in that the same PROC OPTMODEL statements can be applied to any other specific problem of this type (and of any size) by simply changing the data sets appropriately and rerunning the same PROC OPTMODEL statements. Also, because of PROC OPTMODEL's ability to read data very flexibly and from any number of data sets, the problem data can be in its most natural form, making the model easier to explain and understand.

References

IBM (1988). *Mathematical Programming System Extended/370 (MPSX/370) Version 2 Program Reference Manual.* Vol. SH19-6553-0. Armonk, NY: IBM.

Murtagh, B. A. (1981). *Advanced Linear Programming: Computation and Practice*. New York: McGraw-Hill.

Rosenbrock, H. H. (1960). "An Automatic Method for Finding the Greatest or Least Value of a Function." *Computer Journal* 3:175–184.

Chapter 4
Shared Concepts and Topics

Contents

Multithreaded Parallel Computing **21**
Syntax **21**
PERFORMANCE Statement 21
ODS Tables **23**
Memory Limit **23**
Numerical Difficulties **24**
References **25**

Multithreaded Parallel Computing

Although the speed of a single-core processor has increased considerably over the decades, further gains in computing power are possible through the use of multiple cores or processors. This practice is called *parallel computing*, in which certain computations are partitioned into independent smaller subcomputations. Each subcomputation is then processed on separate cores or processors simultaneously. Consumer-grade PCs and servers are often equipped with multicore processors; multiprocessor configurations are becoming relatively common and inexpensive. As a result, parallel computing is becoming increasingly important. One type of parallel computing is multithreaded computing, in which several threads use the processors of a single server to work concurrently on subtasks. These threads share the random access memory (RAM) of that server. In another type of parallel computing, distributed computing, computation is parallelized over several processors (possibly multithreaded), each of which owns an independent memory allocation.

Syntax

PERFORMANCE Statement

PERFORMANCE < *performance-options* > **;**

The PERFORMANCE statement is available in the OPTMODEL, OPTLP, OPTMILP, and OPTQP procedures. This statement can be used to control the parallel execution of multithreaded features such as the concurrent LP algorithm and the OPTMILP option tuner. For an example that demonstrates the use of the PERFORMANCE statement in the OPTMODEL procedure, see Example 10.5 in Chapter 10, "The Nonlinear Programming Solver."

The PERFORMANCE statement is available in both multithreaded and distributed computing environments. This section focuses on the multithreaded computing environment. For information about the PERFORMANCE statement in a distributed computing environment, see Chapter 3, "Shared Concepts and Topics" (*Base SAS Procedures Guide: High-Performance Procedures*).

NOTE: Distributed computing mode requires SAS High-Performance Optimization.

The PERFORMANCE statement enables you to control the number of threads used and the output of the ODS table that reports procedure timing. When you specify the PERFORMANCE statement, the PerformanceInfo ODS table is produced. This table lists performance characteristics such as execution mode and number of threads.

You can specify the following *performance-options* in the PERFORMANCE statement:

DETAILS
requests that the procedure produce the Timing ODS table. This table shows a breakdown of the time used in each step of the procedure.

NTHREADS=*number* **| CPUCOUNT**
specifies the number of threads that a procedure can use. It overrides the SAS system option THREADS | NOTHREADS. The value of *number* can be any integer between 1 and 256 inclusive. The default value is CPUCOUNT, which sets the thread count to the number that is determined by the SAS system option CPUCOUNT=.

Setting the NTHREADS= option to a number greater than the actual number of available cores might result in reduced performance. Specifying a high NTHREADS= value does not guarantee shorter solution time; the actual change in solution time depends on the computing hardware and the scalability of the underlying algorithms in the specified procedure. In some circumstances, a procedure might use fewer threads than the specified value of the NTHREADS= option because the procedure's internal algorithms have determined that a smaller number is preferable.

PARALLELMODE=*string*
specifies the parallel processing mode. This mode determines the solution results that are obtained from running the same model with the same option values on the same platform multiple times.

The values of *string* are listed in Table 4.1.

Table 4.1 Values for PARALLELMODE= Option

string	**Description**
DETERMINISTIC	Requires algorithms to produce the same results every time.
NONDETERMINISTIC	Permits algorithms to produce different solution results. This mode requires less synchronization and might attain better performance than DETERMINISTIC mode.

Some procedures support only one mode; the modes that a procedure supports are detailed in its documentation.

ODS Tables

Anytime you specify the PERFORMANCE statement in a procedure, the procedure generates an ODS table called PerformanceInfo that summarizes the performance characteristics of the procedure. The information comes from the actual characteristics used and does not necessarily match the option values specified in the PERFORMANCE statement. When you specify the DETAILS option in the PERFORMANCE statement, the procedure generates an additional ODS table called Timing.

Output 4.1 shows a typical PerformanceInfo table in multithreaded computing mode.

Figure 4.1 PerformanceInfo Table

The OPTLP Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

If you specify the NOTHREADS system option and do not specify the NTHREADS= option in the PERFORMANCE statement, then the PerformanceInfo table contains the information shown in Output 4.2.

Figure 4.2 PerformanceInfo Table: NOTHREADS Option Specified

The OPTLP Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	Disabled

Output 4.3 demonstrates the contents of a typical Timing table.

Figure 4.3 Timing Table

Procedure Task Timing		
Task	**Time (sec.)**	**Time**
Presolve Time	0.00	0.00%
Solver Time	0.00	0.00%
Wait Time	0.15	100.0%

Memory Limit

The system option MEMSIZE sets a limit on the amount of memory that the SAS System uses. If you do not specify a value for this option, then the SAS System sets a default memory limit. Your operating environment determines the actual size of the default memory limit set by the SAS System, which is sufficient for many applications. However, the solution of many realistic optimization problems can require more memory than the default. It is therefore recommended that the memory limit be increased above the default when you are

solving optimization problems. This reduces the chance of a procedure failing because of an out-of-memory error.

NOTE: The MEMSIZE system option is not available in some operating environments. See the documentation for your operating environment for more information.

You can specify -MEMSIZE 0 to indicate that all available memory can be used, but use this setting with caution. In most operating environments, it is better to specify an adequate amount of memory than to specify -MEMSIZE 0. For example, if you are running PROC OPTLP to solve LP problems with only a few hundred thousand variables and constraints, -MEMSIZE 500M might be sufficient to enable the procedure to run without an out-of-memory error. When a problem has millions of variables, -MEMSIZE 2G or higher might be needed. These are rules of thumb; problems with atypical structure, density, or other characteristics can increase the optimizer's memory requirements.

No matter how much memory is installed, 32-bit Windows operating systems permit the SAS System to use at most 4 gigabytes of memory. This memory limit might be lower, depending on which version of Windows you are running. The limit is enforced by the Windows operating system, not the SAS System.

You can specify the MEMSIZE option at system invocation, on the SAS command line, or in a configuration file. The syntax is described in the *SAS Companion* book for your operating environment.

To report a procedure's memory consumption, you can use the FULLSTIMER option. The syntax is described in the *SAS Companion* book for your operating environment.

Numerical Difficulties

Extremely large or extremely small numerical values might cause computational difficulties (singularities, stalled solution progress, false infeasibilities, and so on) for optimization solvers, but the occurrence of such difficulties is hard to predict. For this reason, solvers issue a data error message when they detect model data that exceed a specific threshold number. The value of the threshold number depends on your operating environment and is printed in the log as part of the data error message.

The following conditions produce a data error:

- The absolute value of an objective coefficient, constraint coefficient, or range (difference between the upper and lower bounds on a constraint) is greater than the threshold number.
- A variable's lower bound, a $\geq$ or = constraint's right-hand side, or a range constraint's lower bound is greater than the threshold number.
- A variable's upper bound, a $\leq$ or = constraint's right-hand side, or a range constraint's upper bound is smaller than the negative threshold number.

If a variable's upper bound is greater than 1E20, then solvers treats the bound as ∞. Similarly, if a variable's lower bound is less than –1E20, then LP solver treats the bound as $-\infty$.

If a solver fails or experiences numerical difficulties when solving a problem, try one of the following remedies:

- Improve the input data: Rescale very large and very small numbers in constraints, objectives, right-hand sides, and variable bounds. It is recommended that the magnitudes of the largest and smallest constraint coefficients not exceed 1E6.
- Specify different algorithms or options (or both): For example, to solve a linear program, you can choose from the primal simplex, dual simplex, interior point, and network simplex algorithms. Using available options, you can tighten or relax feasibility or optimality tolerances.

References

Andrews, G. R. (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, MA: Addison-Wesley.

Chapter 5
The OPTMODEL Procedure

Contents

Overview: OPTMODEL Procedure **28**
Getting Started: OPTMODEL Procedure **29**
An Unconstrained Optimization Example 30
The Rosenbrock Problem 33
A Transportation Problem 34
Syntax: OPTMODEL Procedure **36**
Functional Summary 38
PROC OPTMODEL Statement 40
Declaration Statements 44
Programming Statements 53
Details: OPTMODEL Procedure **95**
Named Parameters 95
Indexing 95
Types 97
Names 98
Parameters 98
Expressions 99
Identifier Expressions 102
Function Expressions 102
Index Sets 103
OPTMODEL Expression Extensions 104
Conditions of Optimality 114
Data Set Input/Output 117
Control Flow 121
Formatted Output 121
ODS Table and Variable Names 124
Constraints 130
Suffixes 134
Integer Variable Suffixes 137
Dual Values 138
Reduced Costs 144
Presolver 145
Model Update 145
Multiple Subproblems 150
Multiple Solutions 151
Problem Symbols 152

OPTMODEL Options 153
Automatic Differentiation 154
Conversions 155
FCMP Routines 156
More on Index Sets 159
Threaded and Distributed Processing 160
Macro Variable _OROPTMODEL_ 161
Rewriting PROC NLP Models for PROC OPTMODEL 163
Examples: OPTMODEL Procedure **166**
Example 5.1: Matrix Square Root 166
Example 5.2: Reading From and Creating a Data Set 167
Example 5.3: Model Construction 169
Example 5.4: Set Manipulation 173
Example 5.5: Multiple Subproblems 174
Example 5.6: Traveling Salesman Problem 178
Example 5.7: Sparse Modeling 182
Example 5.8: Chemical Equilibrium 188
References **192**

Overview: OPTMODEL Procedure

The OPTMODEL procedure includes the powerful OPTMODEL modeling language and state-of-the-art solvers for several classes of mathematical programming problems. The problems and their solvers are listed in Table 5.1.

Table 5.1 Solvers in PROC OPTMODEL

Problem	Solver
Constraint programming	**CLP**
Linear programming	**LP**
Mixed integer linear programming	**MILP**
Network algorithms	**Network**
General nonlinear programming	**NLP**
Quadratic programming	**QP**

The OPTMODEL modeling language provides a modeling environment tailored to building, solving, and maintaining optimization models. This makes the process of translating the symbolic formulation of an optimization model into OPTMODEL virtually transparent since the modeling language mimics the symbolic algebra of the formulation as closely as possible. The OPTMODEL language also streamlines and simplifies the critical process of populating optimization models with data from SAS data sets. All of this transparency produces models that are more easily inspected for completeness and correctness, more easily corrected, and more easily modified, whether through structural changes or through the substitution of new data for old.

In addition to invoking optimization solvers directly with PROC OPTMODEL as already mentioned, you can use the OPTMODEL language purely as a modeling facility. You can save optimization models built with the OPTMODEL language in SAS data sets that can be submitted to other SAS/OR optimization procedures. In general, the OPTMODEL language serves as a common point of access for many of the SAS/OR optimization capabilities, whether providing both modeling and solver access or acting as a modeling interface for other optimization procedures.

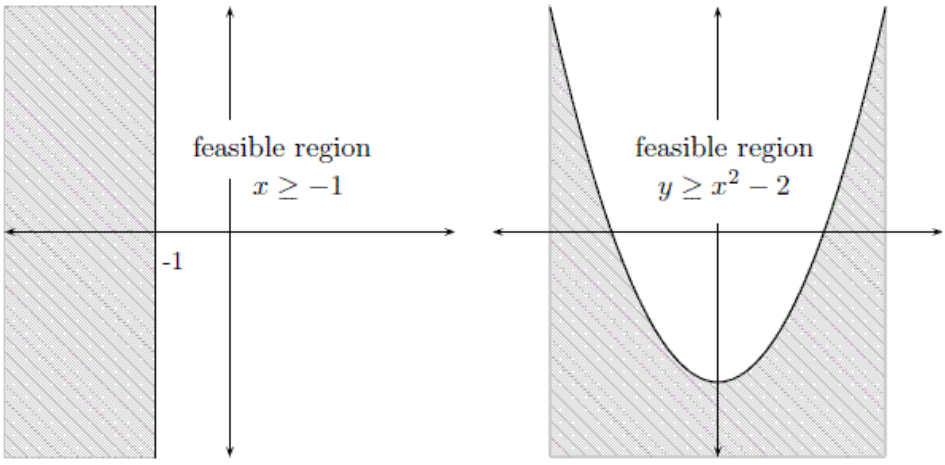
For details and examples of the problems addressed and corresponding solvers, please see the dedicated chapters in this book. This chapter aims to give you a comprehensive understanding of the OPTMODEL procedure by discussing the framework provided by the OPTMODEL modeling language. For additional examples that demonstrate the features of the OPTMODEL procedure, see *SAS/OR User's Guide: Mathematical Programming Examples*.

The OPTMODEL modeling language features automatic differentiation, advanced flow control, optimization-oriented syntax (parameters, variables, arrays, constraints, objective functions), dynamic model generation, model-data separation, and transparent access to SAS data sets.

Getting Started: OPTMODEL Procedure

Optimization or mathematical programming is a search for a maximum or minimum of an *objective function* (also called a *cost function*), where search variables are restricted to particular constraints. Constraints are said to define a *feasible region* (see Figure 5.1).

Figure 5.1 Examples of Feasible Regions



A more rigorous general formulation of such problems is as follows.

Let

$$f : S \rightarrow \mathbb{R}$$

be a real-valued function. Find x^* such that

- $x^* \in S$

- $f(x^*) \leq f(x), \quad \forall x \in S$

Note that the formulation is for the minimum of f and that the maximum of f is simply the negation of the minimum of $-f$.

Here, function f is the *objective function*, and the variable in the objective function is called the optimization variable (or decision variable). S is the *feasible region*. Typically S is a subset of the Euclidean space $\mathbb{R}^n$ specified by the set of *constraints*, which are often a set of equalities ($=$) or inequalities ($\leq, \geq$) that every element in S is required to satisfy simultaneously. For the special case where $S = \mathbb{R}^n$, the problem is an *unconstrained optimization*. An element x of S is called a *feasible solution* to the optimization problem, and the value $f(x)$ is called the *objective value*. A feasible solution x^* that minimizes the objective function is called an *optimal solution* to the optimization problem, and the corresponding objective value is called the *optimal value*.

In mathematics, special notation is used to denote an optimization problem. Generally, you can write an optimization problem as follows:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & x \in S \end{array}$$

Normally, an empty body of constraint (the part after "subject to") implies that the optimization is unconstrained (that is, the feasible region is the whole space $\mathbb{R}^n$). The optimal solution (x^*) is denoted as

$$x^* = \underset{x \in S}{\operatorname{argmin}}\, f(x)$$

The optimal value ($f(x^*)$) is denoted as

$$f(x^*) = \min_{x \in S} f(x)$$

Optimization problems can be classified by the forms (linear, quadratic, nonlinear, and so on) of the functions in the objective and constraints. For example, a problem is said to be *linearly constrained* if the functions in the constraints are linear. A *linear programming* problem is a linearly constrained problem with a linear objective function. A nonlinear programming problem occurs where some function in the objective or constraints is nonlinear, and so on.

An Unconstrained Optimization Example

An unconstrained optimization problem formulation is simply

$$\begin{array}{ll} \text{minimize} & f(x) \end{array}$$

For example, suppose you wanted to find the minimum value of this polynomial:

$$z(x, y) = x^2 - x - 2y - xy + y^2$$

You can compactly specify and solve the optimization problem by using the OPTMODEL modeling language. Here is the program:

```
/* invoke procedure */
proc optmodel;
   var x, y;  /* declare variables */

   /* objective function */
   min z=x**2 - x - 2*y - x*y + y**2;

   /* now run the solver */
   solve;

   print x y;
   quit;
```

This program produces the output in Figure 5.2.

Figure 5.2 Optimizing a Simple Polynomial

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	z
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	0
Constraint Coefficients	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Figure 5.2 *continued*

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	z
Solution Status	Optimal
Objective Value	-2.333333333
Primal Infeasibility	0
Dual Infeasibility	6.861556E-17
Bound Infeasibility	0
Duality Gap	0
Complementarity	0
Iterations	0
Presolve Time	0.00
Solution Time	0.02

x	y
1.3333	1.6667

In PROC OPTMODEL you specify the mathematical formulas that describe the behavior of the optimization problem that you want to solve. In the preceding example there were two independent variables in the polynomial, *x* and *y*. These are the *optimization variables* of the problem. In PROC OPTMODEL you declare optimization variables with the VAR statement. The formula that defines the quantity that you are seeking to optimize is called the *objective function*, or *objective*. The solver varies the values of the optimization variables when searching for an optimal value for the objective.

In the preceding example the objective function is named *z*, declared with the MIN statement. The keyword MIN is an abbreviation for MINIMIZE. The expression that follows the equal sign (=) in the MIN statement defines the function to be minimized in terms of the optimization variables.

The VAR and MIN statements are just two of the many available PROC OPTMODEL declaration and programming statements. PROC OPTMODEL processes all such statements interactively, meaning that each statement is processed as soon as it is complete.

After PROC OPTMODEL has completed processing of declaration and programming statements, it processes the SOLVE statement, which submits the problem to a solver and prints a summary of the results. The PRINT statement displays the optimal values of the optimization variables x and y found by the solver.

It is worth noting that PROC OPTMODEL does not use a RUN statement but instead operates on an interactive basis throughout. You can continue to interact with PROC OPTMODEL even after invoking a solver. For example, you could modify the problem and issue another SOLVE statement (see the section "Model Update" on page 145).

The Rosenbrock Problem

You can use parameters to produce a clear formulation of a problem. Consider the Rosenbrock problem,

$$\text{minimize } f(x_1, x_2) = \alpha\,(x_2 - x_1^2)^2 + (1 - x_1)^2$$

where $\alpha = 100$ is a parameter (constant), x_1 and x_2 are optimization variables (whose values are to be determined), and $f(x_1, x_2)$ is an objective function.

Here is a PROC OPTMODEL program that solves the Rosenbrock problem:

```
proc optmodel;
   number alpha = 100; /* declare parameter */
   var x {1..2};       /* declare variables */
   /* objective function */
   min f = alpha*(x[2] - x[1]**2)**2 +
           (1 - x[1])**2;
   /* now run the solver */
   solve;

   print x;
   quit;
```

The PROC OPTMODEL output is shown in Figure 5.3.

Figure 5.3 Rosenbrock Function Results

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Nonlinear
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Figure 5.3 *continued*

Solution Summary	
Solver	NLP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	8.204873E-23
Optimality Error	9.704881E-11
Infeasibility	0
Iterations	14
Presolve Time	0.00
Solution Time	0.00

[1]	x
1	1
2	1

A Transportation Problem

You can easily translate the symbolic formulation of a problem into the OPTMODEL procedure. Consider the transportation problem, which is mathematically modeled as the following linear programming problem:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i \in O, j \in D} c_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{j \in D} x_{ij} = a_i, \quad \forall i \in O \qquad \text{(SUPPLY)} \\
& \sum_{i \in O} x_{ij} = b_j, \quad \forall j \in D \qquad \text{(DEMAND)} \\
& x_{ij} \geq 0, \quad \forall (i,j) \in O \times D
\end{aligned}$$

where O is the set of origins, D is the set of destinations, c_{ij} is the cost to transport one unit from i to j, a_i is the supply of origin i, b_j is the demand of destination j, and x_{ij} is the decision variable for the amount of shipment from i to j.

Here is a very simple example. The cities in the set O of origins are Detroit and Pittsburgh. The cities in the set D of destinations are Boston and New York. The cost matrix, supply, and demand are shown in Table 5.2.

Table 5.2 A Transportation Problem

	Boston	New York	Supply
Detroit	30	20	200
Pittsburgh	40	10	100
Demand	150	150	

The problem is compactly and clearly formulated and solved by using the OPTMODEL procedure with the following statements:

```
proc optmodel;
   /* specify parameters */
   set O={'Detroit','Pittsburgh'};
   set D={'Boston','New York'};
   number c{O,D}=[30 20
                  40 10];
   number a{O}=[200 100];
   number b{D}=[150 150];
   /* model description */
   var x{O,D} >= 0;
   min total_cost = sum{i in O, j in D}c[i,j]*x[i,j];
   constraint supply{i in O}: sum{j in D}x[i,j]=a[i];
   constraint demand{j in D}: sum{i in O}x[i,j]=b[j];
   /* solve and output */
   solve;
   print x;
```

The output is shown in Figure 5.4.

Figure 5.4 Solution to the Transportation Problem

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	total_cost
Objective Type	Linear
Number of Variables	4
Bounded Above	0
Bounded Below	4
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	4
Linear LE (<=)	0
Linear EQ (=)	4
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	8

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Figure 5.4 *continued*

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	total_cost
Solution Status	Optimal
Objective Value	6500
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	0
Presolve Time	0.00
Solution Time	0.00

x		
	Boston	**New York**
Detroit	150	50
Pittsburgh	0	100

Syntax: OPTMODEL Procedure

PROC OPTMODEL statements are divided into three categories: the PROC statement, the declaration statements, and the programming statements. The PROC statement invokes the procedure and sets initial option values. The declaration statements declare optimization model components. The programming statements read and write data, invoke the solver, and print results. In the following text, the statements are listed in the order in which they are grouped, with declaration statements first.

NOTE: Solver specific options are described in the individual chapters that correspond to the solvers.

PROC OPTMODEL *options* **;**

Declaration Statements:
CONSTRAINT *constraints* **;**
IMPVAR *optimization expression declarations* **;**
MAX *objective* **;**
MIN *objective* **;**
NUMBER *parameter declarations* **;**
PROBLEM *problem declaration* **;**
SET [**<** *types* **>**] *parameter declarations* **;**
STRING *parameter declarations* **;**
VAR *variable declarations* **;**

Programming Statements:
Assignment parameter = *expression* **;**
CALL *name* [**(** *expressions* **)**] **;**
CLOSEFILE *files* **;**
COFOR { *index-set* **}** *statement* **;**
CONTINUE ;
CREATE DATA *SAS-data-set* **FROM** *columns* **;**
DO ; *statements* **; END ;**
DO *variable* **=** *specifications* **;** *statements* **; END ;**
DO UNTIL (*logic* **) ;** *statements* **; END ;**
DO WHILE (*logic* **) ;** *statements* **; END ;**
DROP *constraint* **;**
EXPAND *name* [/ *options*] **;**
FILE *file* **;**
FIX *variable* [**=** *expression*] **;**
FOR { *index-set* **}** *statement* **;**
IF *logic* **THEN** *statement* **;** [**ELSE** *statement*] **;**
LEAVE ;
(null statement) **;**
PERFORMANCE *options* **;**
PRINT *print items* **;**
PROFILE [*mode*] *options* **;**
PUT *put items* **;**
QUIT ;
READ DATA *SAS-data-set* **INTO** *columns* **;**
RESET OPTIONS *options* **;**
RESTORE *constraint* **;**
SAVE MPS *SAS-data-set* [**OBJECTIVE** *name*] [**NOOBJECTIVE**] **;**
SAVE QPS *SAS-data-set* [**OBJECTIVE** *name*] [**NOOBJECTIVE**] **;**
SOLVE [**WITH** *solver*] [**OBJECTIVE** *name*] [**NOOBJECTIVE**] [**RELAXINT**] [/ *options*] **;**
STOP ;
SUBMIT *arguments* [/ *options*] **;**
UNFIX *variable* [**=** *expression*] **;**
USE PROBLEM *problem* **;**

Functional Summary

The statements and options available with PROC OPTMODEL are summarized by purpose in Table 5.3.

Table 5.3 Functional Summary

Description	Statement	Option
Declaration Statements:		
Declares a constraint	CONSTRAINT	
Declares optimization expressions	IMPVAR	
Declares a maximization objective	MAX	
Declares a minimization objective	MIN	
Declares a number type parameter	NUMBER	
Declares a problem	PROBLEM	
Declares a set type parameter	SET	
Declares a string type parameter	STRING	
Declares optimization variables	VAR	
Programming Statements:		
Assigns a value to a variable or parameter	=	
Invokes a library subroutine	CALL	
Closes the opened file	CLOSEFILE	
Executes the statement repeatedly with support for concurrent solver invocations	COFOR	
Terminates one iteration of a loop statement	CONTINUE	
Creates a new SAS data set and copies data into it from PROC OPTMODEL parameters and variables	CREATE DATA	
Groups a sequence of statements together as a single statement	DO	
Executes statements repeatedly	DO (iterative)	
Executes statements repeatedly until some condition is satisfied	DO UNTIL	
Executes statements repeatedly as long as some condition is satisfied	DO WHILE	
Ignores the specified constraint	DROP	
Prints the specified constraint, variable, or objective declaration expressions after expanding aggregation operators, and so on	EXPAND	
Selects a file for the PUT statement	FILE	
Treats a variable as fixed in value	FIX	
Executes the statement repeatedly	FOR	
Executes the statement conditionally	IF	
Terminates the execution of the entire loop body	LEAVE	
Null statement	;	
Controls parallel execution	PERFORMANCE	

Description	Statement	Option
Outputs string and numeric data	PRINT	
Provides timing and execution count information for statements and declarations	PROFILE	
Writes text data to the current output file	PUT	
Terminates the PROC OPTMODEL session	QUIT	
Reads data from a SAS data set into PROC OPTMODEL parameters and variables	READ DATA	
Sets PROC OPTMODEL option values or restores them to their defaults	RESET OPTIONS	
Adds a constraint that was previously dropped back into the model	RESTORE	
Saves the structure and coefficients for a linear programming model into a SAS data set	SAVE MPS	
Saves the structure and coefficients for a quadratic programming model into a SAS data set	SAVE QPS	
Invokes a PROC OPTMODEL solver	SOLVE	
Halts the execution of all statements that contain it	STOP	
Submits SAS code for execution	SUBMIT	
Reverses the effect of FIX statement	UNFIX	
Selects the current problem	USE PROBLEM	
PROC OPTMODEL Options:		
Specifies the accuracy for nonlinear constraints	PROC OPTMODEL	CDIGITS=
Specifies the maximum number of error messages displayed	PROC OPTMODEL	ERRORLIMIT=
Specifies the method used to approximate numeric derivatives	PROC OPTMODEL	FD=
Specifies the accuracy for the objective function	PROC OPTMODEL	FDIGITS=
Forces finite differences to be used for nonlinear equations	PROC OPTMODEL	FORCEFD=
Enables the OPTMODEL presolver for the CLP, LP, MILP, and QP solvers	PROC OPTMODEL	FORCEPRESOLVE=
Passes initial values for variables to the solver	PROC OPTMODEL	INITVAR/NOINITVAR
Specifies the tolerance for rounding the bounds on integer and binary variables	PROC OPTMODEL	INTFUZZ=
Specifies the maximum length for MPS row and column labels	PROC OPTMODEL	MAXLABLEN=
Checks missing values	PROC OPTMODEL	MISSCHECK/NOMISSCHECK
Specifies the maximum number of non-error messages displayed	PROC OPTMODEL	MSGLIMIT=
Specifies the number of digits to display	PROC OPTMODEL	PDIGITS=
Adjusts how two-dimensional array is displayed	PROC OPTMODEL	PMATRIX=
Specifies the type of presolve performed by the PROC OPTMODEL presolver	PROC OPTMODEL	PRESOLVER=

Description	Statement	Option
Specifies the tolerance, enabling the PROC OPTMODEL presolver to remove slightly infeasible constraints	PROC OPTMODEL	PRESTOL=
Enables or disables printing summary	PROC OPTMODEL	PRINTLEVEL=
Specifies the width to display numeric columns	PROC OPTMODEL	PWIDTH=
Specifies the smallest difference that is permitted by the PROC OPTMODEL presolver between the upper and lower bounds of an unfixed variable	PROC OPTMODEL	VARFUZZ=

PROC OPTMODEL Statement

PROC OPTMODEL [*options*] **;**

The PROC OPTMODEL statement invokes the OPTMODEL procedure. You can specify options to control how the optimization model is processed and how results are displayed. You can specify the following *options* (these options can also be specified in the RESET OPTIONS statement).

CDIGITS=*number*

specifies the expected number of decimal digits of accuracy for nonlinear constraints. The value can be fractional. PROC OPTMODEL uses this option to choose a step length when numeric derivative approximations are required to evaluate the Jacobian of nonlinear constraints. The default value depends on your operating environment. It is assumed that constraint values are accurate to the limits of machine precision.

See the section "Automatic Differentiation" on page 154 for more information about numeric derivative approximations.

ERRORLIMIT=*number* | **NONE**

specifies the maximum number of error messages that can be displayed during SOLVE statement processing. Specifying a value of *number* in the range 1 to $2^{31} - 1$ sets a specific limit. Specifying ERRORLIMIT=NONE removes any existing limit. The default value is 10.

NOTE: Some errors abort processing immediately.

FD=FORWARD | CENTRAL

selects the method used to approximate numeric derivatives when analytic derivatives are unavailable. Most solvers require the derivatives of the objective and constraints. You can specify the following values:

FORWARD uses forward differences.

CENTRAL uses central differences.

By default, FD=FORWARD. For more information about numeric derivative approximations, see the section "Automatic Differentiation" on page 154.

FDIGITS=*number*

specifies the expected number of decimal digits of accuracy for the objective function. The value can be fractional. PROC OPTMODEL uses the value to choose a step length when numeric derivatives are required. The default value depends on your operating environment. It is assumed that objective function values are accurate to the limits of machine precision.

For more information about numeric derivative approximations, see the section "Automatic Differentiation" on page 154.

FORCEFD=NONE | OBJ | CON | ALL

forces PROC OPTMODEL to use finite differences instead of analytic derivatives for the specified set of nonlinear expressions. This option can be useful with FCMP functions to provide more control over derivative computation. You can specify the following values:

ALL restricts all derivative computations to use finite differences.

CON restricts derivative computations for the nonlinear constraint expressions and any IMPVAR expressions they reference to use finite differences.

NONE requests analytic derivatives where they are available.

OBJ restricts derivative computations for the objective and any IMPVAR expressions it references to use finite differences.

By default, FORCEFD=NONE.

FORCEPRESOLVE=*number* | *string*

specifies whether PROC OPTMODEL can use the OPTMODEL presolver with the CLP, LP, MILP, and QP solvers. By default, the OPTMODEL presolver is disabled when PROC OPTMODEL solves linear problems or problems with predicates, or when the CLP, LP, MILP, or QP solver is specified in the SOLVE statement. Table 5.4 shows the valid values for this option.

Table 5.4 Values for the FORCEPRESOLVE= Option

number	*string*	**Description**
0	OFF	Restores the default behavior.
1	ON	Enables PROC OPTMODEL to use the OPTMODEL presolver when the CLP, LP, MILP, or QP solver is specified in the SOLVE statement.

By default, FORCEPRESOLVE=0.

INITVAR | NOINITVAR

selects whether or not to pass initial values for variables to the solver when the SOLVE statement is executed. INITVAR enables the current variable values to be passed. NOINITVAR causes the solver to be invoked without any specific initial values for variables. The INITVAR option is the default.

The CLP, LP, and QP solvers always ignore initial values. The NLP solvers attempt to use specified initial values. The MILP solver uses initial values only if the PRIMALIN option is specified.

INTFUZZ=*number*
specifies the tolerance for rounding the bounds on integer and binary variables to integer values. Bounds that differ from an integer by at most *number* are rounded to that integer. Otherwise, lower bounds are rounded up to the next greater integer and upper bounds are rounded down to the next lesser integer. The value of *number* can range between 0 and 0.5. The default value is 0.00001.

MAXLABLEN=*number*
specifies the maximum length for MPS row and column labels. The allowed range is 8 to 256. This option can also be used to control the length of row and column names displayed by solvers, such as those found in the LP solver iteration log. See also the description of the .label suffix in the section "Suffixes" on page 134. By default, MAXLABLEN=32.

MISSCHECK | NOMISSCHECK
enables detailed checking of missing values in expressions. MISSCHECK requests that PROC OPTMODEL produce a message each time it evaluates an arithmetic operation or function that has missing value operands (except when the operation or function specifically supports missing values). The MISSCHECK option can increase processing time. NOMISSCHECK turns off this detailed reporting. NOMISSCHECK is the default.

MSGLIMIT=*number* **| NONE**
specifies the maximum number of messages about certain issues that can be displayed during processing of a single top-level statement, such as a SOLVE or FOR statement. Specifying a value of *number* in the range 0 to $2^{31} - 1$ sets a specific limit. Specifying MSGLIMIT=NONE removes any existing limit. The default value is 25.

The limit is applied to notes and warnings for the following issues:

- arithmetic evaluation issues, such as division by zero
- function evaluation issues, such as invalid arguments
- problem generation and PROC OPTMODEL presolver issues
- duplicate members in set constructor and literal expressions
- string concatenation results truncated to the maximum string length
- duplicate READ DATA keys
- truncated data set column labels for the CREATE DATA statement
- misspelled keywords for an option value specified using a string expression
- unrecognized file specifications in the CLOSEFILE statement

NOTE: Because of complications of concurrent execution, PROC OPTMODEL might display more or fewer messages than the limit when you use a COFOR statement.

PDIGITS=*number*
requests that the PRINT statement display *number* significant digitsfor numeric columns for which no format is specified. The value can range from 1 to 9. By default, PDIGITS=5.

PMATRIX=*number*
adjusts the density evaluation of a two-dimensional array to affect how it is displayed. The value *number* scales the total number of nonempty array elements and is used by the PRINT statement to evaluate whether a two-dimensional array is "sparse" or "dense." Tables that contain a single

two-dimensional array are printed in list form if they are sparse and in matrix form if they are dense. Any nonnegative value can be assigned to *number*. Specifying a value for the PMATRIX= option that is less than 1 causes the list form to be used in more cases, whereas specifying a value greater than 1 causes the matrix form to be used in more cases. If the value is 0, then the list form is always used. For more information, see the section "PRINT Statement" on page 75. By default, PMATRIX=1.

PRESOLVER=*number* | *string*

specifies the type of presolve that the OPTMODEL presolver performs. Table 5.5 shows the valid values of this option.

Table 5.5 Values for the PRESOLVER= Option

number	*string*	**Description**
–1	AUTOMATIC	Applies presolver using default setting.
0	NONE	Disables presolver.
1	BASIC	Performs minimal processing, only substituting fixed variables and removing empty feasible constraints.
2	MODERATE	Applies a higher level of presolve processing.
3	AGGRESSIVE	Applies the highest level of presolve processing.

The OPTMODEL presolver tightens variable bounds and eliminates redundant constraints. In general, this tightening improves the performance of any solver. Higher levels of presolve processing allow more tightening and substitution passes, but might take more time to execute. The AUTOMATIC option is intermediate between the MODERATE and AGGRESSIVE levels.

NOTE: The OPTMODEL presolver is normally bypassed when PROC OPTMODEL uses the CLP, LP, QP, MILP, or network solver and when the SAVE MPS and SAVE QPS statements execute. The FORCEPRESOLVE= option enables the OPTMODEL presolver to be used with the CLP, LP, QP, and MILP solvers. PROC OPTMODEL always bypasses the OPTMODEL presolver when you specify certain solver options. For more information, see the chapter for the relevant solver in this book.

PRESTOL=*number*

provides a tolerance so that slightly infeasible constraints can be eliminated by the OPTMODEL presolver. If the magnitude of the infeasibility is no greater than num($|X| + 1$), where X is the value of the original bound, then the empty constraint is removed from the presolved problem. OPTMODEL's presolver does not print messages about infeasible constraints and variable bounds when the infeasibility is within the PRESTOL tolerance. The value of PRESTOL can range between 0 and 0.1; the default value is 1E–12.

PRINTLEVEL=*number*

controls the level of listing output during a SOLVE or COFOR command. The Output Delivery System (ODS) tables printed at each level are listed in Table 5.6. Some solvers can produce additional tables; see the individual solver chapters for more information.

Table 5.6 Values for the PRINTLEVEL= Option

number	**Description**
0	Disables all tables.

number	**Description**
1	Prints COFOR Performance Information, Problem Summary, Performance Information, and Solution Summary.
2	Prints COFOR Performance Information, Problem Summary, Performance Information, Solution Summary, Methods of Derivative Computation (for NLP solvers), Solver Options, Optimization Statistics, and solver-specific ODS tables.

For more information about the ODS tables produced by PROC OPTMODEL, see the section "ODS Table and Variable Names" on page 124.

PWIDTH=*number*

sets the width used by the PRINT statement to display numeric columns when no format is specified. The smallest value *number* can take is the value of the PDIGITS= option plus 7; the largest value *number* can take is 16. The default value is equal to the value of the PDIGITS= option plus 7.

VARFUZZ=*number*

specifies the smallest difference that is permitted by the OPTMODEL presolver between the upper and lower bounds of an unfixed variable. If the difference is smaller than *number*, then the variable isfixed to the average of the upper and lower bounds before it is presented to the solver. Any nonnegative value can be assigned to *number*; the default value is 0.

Declaration Statements

The declaration statements define the parameters, variables, constraints, and objectives that describe a PROC OPTMODEL optimization model. Declarations in the PROC OPTMODEL input are saved for later use. Unlike programming statements, declarations cannot be nested in other statements. Declaration statements are terminated by a semicolon.

Many declaration attributes, such as variable bounds, are defined using expressions. Expressions in declarations are handled symbolically and are resolved as needed. In particular, expressions are generally reevaluated when one of the parameter values they use has been changed.

CONSTRAINT Declaration

CONSTRAINT *constraint* [**,** *. . . constraint*] **;**

CON *constraint* [**,** *. . . constraint*] **;**

The constraint declaration defines one or more constraints on expressions in terms of the optimization variables. You can specify multiple constraint declaration statements.

Constraints can have an upper bound, a lower bound, or both bounds. The allowed forms are as follows:

[*name* [**{** *index-set* **}**] **:**] *expression* **=** *expression*

declares an equality constraint or, when an *index-set* is specified, a family of equality constraints. The solver attempts to assign values to the optimization variables to make the two expressions equal.

[*name* [**{** *index-set* **}**] **:**] *expression* ~**=** *expression*

declares a disequality constraint or, when an *index-set* is specified, a family of disequality constraints. The solver attempts to assign values to the optimization variables to make the two expressions unequal. The CLP solver must be used with this type of constraint.

[*name* [**{** *index-set* **}**] **:**] *expression relation expression*

declares an inequality constraint that has a single upper or lower bound. *index-set* declares a family of inequality constraints. *relation* is the <=, <, >=, or > operator. When *relation* is the <= operator, the solver tries to assign optimization variable values so that the value of the left *expression* is less than or equal to the value of the right *expression*. When *relation* is the < operator, the solver tries to assign optimization variable values so that the value of the left *expression* is less than the value of the right *expression*. When *relation* is the >= operator, the solver tries to assign optimization variable values so that the value of the left *expression* is greater than or equal to the value of the right *expression*. When *relation* is the > operator, the solver tries to assign optimization variable values so that the value of the left *expression* is greater than the value of the right *expression*. The CLP solver must be used when the < or > operator is specified.

[*name* [**{** *index-set* **}**] **:**] *bound relation body relation bound*

declares an inequality constraint that is bounded on both sides, called a range constraint. *index-set* declares a family of range constraints. *relation* is the <=, <, >=, or > operator. Both *relation* operators must match in direction. If the <= or < operator is used in the first position, then a <= or < operator must be used in the second position. If the >= or > operator is used in the first position, then a >= or > operator must be used in the second position. The first *bound* expression defines the lower bound (if the <= or < operator is used) or the upper bound (if the >= or > operator is used). The second *bound* defines the upper bound (if the <= or < operator is used) or the lower bound (if the >= or > operator is used). The solver tries to assign optimization variables so that the value of the *body* expression is in the range between the upper and lower bounds. The CLP solver must be used when the < or > operator is specified.

[*name* [**{** *index-set* **}**] **:**] *predicate*

declares a predicate constraint for the CLP solver. See the section "Predicates" on page 202 in Chapter 6, "The Constraint Programming Solver," for a description of the syntax and meaning of *predicates*.

NOTE: You can use the alternate forms from Table 5.10 for the relational operators.

name defines the name for the constraint. Use the name to reference constraint attributes, such as the bounds, elsewhere in the PROC OPTMODEL model. If no name is provided, then a default name is created of the form _ACON_[*n*], where *n* is an integer. See the section "Constraints" on page 130 for more information.

Here is a simple example that defines a constraint with a lower bound:

```
proc optmodel;
   var x, y;
   number low;
   con a: x+y >= low;
```

The following example adds an upper bound:

```
var x, y;
number low;
con a: low <= x+y <= low+10;
```

Indexed families of constraints can be defined by specifying an *index-set* after the name. Any dummy parameters that are declared in the *index-set* can be referenced in the expressions that define the constraint. A particular member of an indexed family can be specified by using an *identifier-expression* with a bracketed index list, in the same fashion as array parameters and variables. For example, the following statements create an indexed family of constraints named incr:

```
proc optmodel;
   number n;
   var x{1..n}
   /* require nondecreasing x values */
   con incr{i in 1..n-1}: x[i+1] >= x[i];
```

The CON statement in the example creates constraints incr[1] through incr[n–1].

Constraint expressions cannot be defined using functions that return different values each time they are called. See the section "Indexing" on page 95 for details.

IMPVAR Declaration

IMPVAR *impvar-decl* [**,** *. . . impvar-decl*] **;**

The IMPVAR declaration specifies one or more names that refer to optimization expressions in the model. The declared name is called an implicit variable. An implicit variable is useful for structuring models so that complex expressions do not need to be repeated each time they are used. The value of an implicit variable needs to be computed only once instead of at each place where the original expression is used, which helps reduce computational overhead. Implicit variables are evaluated without intervention from the solver.

Multiple IMPVAR declarations are allowed. The names of implicit variables must be distinct from other model declarations, such as variables and constraints. Implicit variables can be used in model expressions in the same places where ordinary variables are allowed.

This is the syntax for an *impvar-decl*:

name [**{** *index-set* **}**] **=** *expression*

Each *impvar-decl* declares a name for an implicit variable. The name can be followed by an *index-set* specification to declare a family of implicit variables. The *expression* that the name refers to follows. Dummy parameters that are declared in the *index-set* specification can be used in the expression. The *expression* can refer to other model components, including variables, the current implicit variable, and other implicit variables.

As an example, in the following model statements the implicit variable total_weight is used in multiple constraints to set a limit on various product quantities, represented by locations in array x:

```
impvar total_weight = sum{p in PRODUCTS} Weight[p]*x[p];

con prod1_limit: Weight['Prod1'] * x['Prod1'] <= 0.3 * total_weight;
con prod2_limit: Weight['Prod2'] * x['Prod2'] <= 0.25 * total_weight;
```

MAX and MIN Objective Declarations

MAX *name* [**{** *index-set* **}**] **=** *expression* **;**

MIN *name* [**{** *index-set* **}**] **=** *expression* **;**

The MAX or MIN declaration specifies an objective for the solver. The *name* names the objective function for later reference. When a non-array objective declaration is read, the declaration becomes the new objective of the current problem, replacing any previous objective. The solver maximizes an objective that is specified with the MAX keyword and minimizes an objective that is specified with the MIN keyword. An objective is not allowed to have the same name as a parameter or variable. Multiple objectives are permitted, but the solver processes only one objective at a time.

expression specifies the numeric function to maximize or minimize in terms of the optimization-variables. Specify an *index-set* to declare a family of objectives. Dummy parameters declared in the *index-set* specification can be used in the following expression.

Objectives can also be used as implicit variables. When used in an expression, an objective name refers to the current value of the named objective function. The value of an unsuffixed objective name can depend on the value of optimization variables, so objective names cannot be used in constant expressions such as variable bounds. You can reference objective names in objective or constraint expressions. For example, the following statements declare two objective names, q and l, which are immediately referred to in the objective declaration of z and the declarations of the constraints.

```
proc optmodel;
   var x, y;
   min q=(x+y)**2;
   max l=x+2*y;
   min z=q+l;
   con c1: q<=4;
   con c2: l>=2;
```

Objectives cannot be defined using functions that return different values each time they are called. See the section "Indexing" on page 95 for details.

NUMBER, STRING, and SET Parameter Declarations

NUMBER *parameter-decl* [**,** *. . . parameter-decl*] **;**

STRING *parameter-decl* [**,** *. . . parameter-decl*] **;**

SET [**<** *scalar-type*, *. . . scalar-type* **>**] *parameter-decl* [**,** *. . . parameter-decl*] **;**

Parameters provide names for constants. Parameters are declared by specifying the parameter type followed by a list of parameter names. Declarations of parameters that have NUMBER or STRING types start with a *scalar-type* specification:

NUMBER | **NUM** **;**

STRING | **STR** **;**

The NUM and STR keywords are abbreviations for the NUMBER and STRING keywords, respectively.

The declaration of a parameter that has the set type begins with a *set-type* specification:

SET [**<** *scalar-type*, *. . . scalar-type* **>**] **;**

In a *set-type* declaration, the SET keyword is followed by a list of *scalar-type* items that specify the member type. A set with scalar members is specified with a single *scalar-type* item. A set with tuple members has a *scalar-type* item for each tuple element. The *scalar-type* items specify the types of the elements at each tuple position.

If the SET keyword is not followed by a list of *scalar-type* items, then the set type is determined from the type of the initialization expression. The declared type defaults to SET<NUMBER> if no initialization expression is given or if the expression type cannot be determined.

For any parameter type, the type declaration is followed by a list of *parameter-decl* items that specify the names of the parameters to declare. In a *parameter-decl* item the parameter name can be followed by an optional index specification and any necessary options, as follows:

name [**{** *index-set* **}**] [*parameter-options*]

The parameter *name* and *index-set* can be followed by a list of *parameter-options*. Dummy parameters declared in the *index-set* can be used in the *parameter-options*. The parameter options can be specified with the following forms:

= *expression*
: provides an explicit value for each parameter location. In this case the parameter acts like an alias for the *expression* value.

INIT *expression*
: specifies a default value that is used when a parameter value is required but no other value has been supplied. For example:

```
number n init 1;
set s init {'a', 'b', 'c'};
```

PROC OPTMODEL evaluates the expression for each parameter location the first time the parameter needs to be resolved. The expression is not used when the parameter already has a value.

= [*initializers* **]**
: provides a compact means to define the values for an array, in which each array location value can be individually specified by the *initializers*.

INIT [*initializers* **]**
: provides a compact means to define multiple default values for an array. Each array location value can be individually specified by the *initializers*. With this option the array values can still be updated outside the declaration.

The =*expression* parameter option defines a parameter value by using a formula. The formula can refer to other parameters. The parameter value is updated when the referenced parameters change. The following example shows the effects of the update:

```
proc optmodel;
   number n;
   set<number> s = 1..n;
   number a{s};
   n = 3;
   a[1] = 2;    /* OK */
```

```
a[7] = 19;   /* error, 7 is not in s */
n = 10;
a[7] = 19;   /* OK now */
```

In the preceding example the value of set s is resolved for each use of array a that has an index. For the first use of a[7], the value 7 is not a member of the set s. However, the value 7 is a member of s at the second use of a[7].

The INIT *expression* parameter option specifies a default value for a parameter. The following example shows the usage of this option:

```
proc optmodel;
   num a{i in 1..2} init i**2;
   a[1] = 2;
   put a[*]=;
```

When the value of a parameter is needed but no other value has been supplied, the default value specified by INIT *expression* is used, as shown in Figure 5.5.

Figure 5.5 INIT Option: Output

```
a[1]=2 a[2]=4
```

NOTE: Parameter values can also be read from files or specified with assignment statements. However, the value of a parameter that is assigned with the =*expression* or =[*initializers*] forms can be changed only by modifying the parameters used in the defining expressions. Parameter values specified by the INIT option can be reassigned freely.

Initializing Arrays

Arrays can be initialized with the =[*initializers*] or INIT [*initializers*] forms. These forms are convenient when array location values need to be individually specified. The forms behave the same way, except that the INIT [*initializers*] form allows the array values to be modified after the declaration. These forms of initialization are used in the following statements:

```
proc optmodel;
   number a{1..3} = [5 4 7];
   number b{1..3} INIT [5 4 7];
   put a[*]=;
   b[1] = 1;
   put b[*]=;
```

Each array location receives a different value, as shown in Figure 5.6. The displayed values for b are a combination of the default values from the declaration and the assigned value in the statements.

Figure 5.6 Array Initialization

```
a[1]=5 a[2]=4 a[3]=7
b[1]=1 b[2]=4 b[3]=7
```

Each *initializer* takes the following form:

[**[** *index* **]**] *value*

The *value* specifies the value of an array location and can be a numeric or string constant, a set literal, or an expression enclosed in parentheses.

In array initializers, string constants can be specified using quoted strings. When the string text follows the rules for a SAS name, the text can also be specified without quotation marks. String constants that begin with a digit, contain blanks, or contain other special characters must be specified with a quoted string.

As an example, the following statements define an array parameter that could be used to map numeric days of the week to text strings:

```
proc optmodel;
   string dn{1..5} =
          [Monday Tuesday Wednesday Thursday Friday];
```

The optional *index* in square brackets specifies the index of the array location to initialize. The index specifies one or more numeric or string subscripts. The subscripts allow the same syntactic forms as the *value* items. Commas can be used to separate index subscripts. For example, location a[1,'abc'] of an array a could be specified with the index [1 abc]. The following example initializes just the diagonal locations in a square array:

```
proc optmodel;
   number m{1..3,1..3} = [[1 1] 0.1 [2 2] 0.2 [3 3] 0.3];
```

An index does not need to specify all the subscripts of an array location. If the index begins with a comma, then only the rightmost subscripts of the index need to be specified. The preceding subscripts are supplied from the index that was used by the preceding *initializer*. This can simplify the initialization of arrays that are indexed by multiple subscripts. For example, you can add new entries to the matrix of the previous example by using the following statements:

```
proc optmodel;
   number m{1..3,1..3} = [[1 1] 0.1            [,3] 1
                                  [2 2] 0.2    [,3] 2
                                             [3 3] 0.3];
```

The spacing shows the layout of the example array. The previous example was updated by initializing two more values at m[1,3] and m[2,3].

If an index is omitted, then the next location in the order of the array's index set is initialized. If the index set has multiple *index-set-items*, then the rightmost indices are updated before indices to the left are updated. At the beginning of the initializer list, the rightmost index is the first member of the index set. The index set must use a range expression to avoid unpredictable results when an index value is omitted.

The initializers can be followed by commas. The use of commas has no effect on the initialization. The comma can be used to clarify layout. For example, the comma could separate rows in a matrix.

Not every array location needs to be initialized. The locations without an explicit initializer are set to zero for numeric arrays, set to an empty string for string arrays, and set to an empty set for set arrays.

NOTE: An array location must not be initialized more than once during the processing of the initializer list.

PROBLEM Declaration

PROBLEM *name* [**{** *index-set* **}**] [**FROM** *problem-id*] [**INCLUDE** *problem-items*] **;**

Problems are declared with the PROBLEM declaration. Problem declarations track an objective, a set of included variables and constraints, and some status information that is associated with the variables and constraints. The problem name can optionally be followed by an *index-set* to create a family of problems. When a problem is first used (via the USE PROBLEM statement), the specifications from the optional FROM and INCLUDE clauses create the initial set of included variables, constraints, and the problem objective. An empty problem is created if neither clause is specified. The clauses are applied only when the problem is first used with the USE PROBLEM statement.

The FROM clause specifies an existing problem from which to copy the set of included symbols. The *problem-id* is an *identifier expression*. The dropped and fixed status for these symbols in the specified problem is also copied.

The INCLUDE clause specifies a list of variables, constraints, and objectives to include in the problem. These items are included with default status (unfixed and undropped) which overrides the status from the FROM clause, if it exists. Each item is specified with one of the following forms:

identifier-expression
: includes the specified items in the problem. The *identifier-expression* can be a symbol name or an array symbol with explicit index. If an array symbol is used without an index, then all array elements are included.

{ *index-set* **}** *identifier-expression*
: includes the specified subset of items in the problem. The item specified by the *identifier-expression* is added to the problem for each member of the *index-set*. The dummy parameters from the *index-set* can be used in the indexing of the *identifier-expression*. If the *identifier-expression* is an array symbol without indexing, then the *index-set* provides the indices for the included locations.

You can use the FROM and INCLUDE clauses to designate the initial objective for a problem. The objective is copied from the problem designated by the FROM clause, if present. Then the INCLUDE clause, if any, is applied, and the last objective specified becomes the initial objective.

The following statements declare some problems with a variable x and different objectives to illustrate some of the ways of including model components. Note that the statements use the predeclared problem _START_ to avoid resetting the objective in prob2 when the objective z3 is declared.

```
proc optmodel;
   problem prob1;
   use problem prob1;
   var x >= 0;              /* included in prob1 */
   min z1 = (x-1)**2;       /* included in prob1 */
   expand;                  /* prob1 contains x, z1 */

   problem prob2 from prob1;
   use problem prob2;       /* includes x, z1 */
   min z2 = (x-2)**2;       /* resets prob2 objective to z2 */
   expand;                  /* prob2 contains x, z2 */

   use problem _start_;     /* don't modify prob2 */
```

```
min z3 = (x-3)**2;
problem prob3 include x z3;
use problem prob3;
expand;                  /* prob3 contains x, z3 */
```

See the section "Multiple Subproblems" on page 150 for more details about problem processing.

VAR Declaration

VAR *var-decl* [**,** *. . . var-decl*] **;**

The VAR statement declares one or more optimization variables. Multiple VAR statements are permitted. A variable is not allowed to have the same name as a parameter or constraint.

Each *var-decl* specifies a variable name. The name can be followed by an array *index-set* specification and then variable options. Dummy parameters declared in the index set specification can be used in the following variable options.

Here is the syntax for a *var-decl*:

name [**{** *index-set* **}**] [*var-options*]

For example, the following statements declare a group of 100 variables, x[1]–x[100]:

```
proc optmodel;
   var x{1..100};
```

Here are the available variable options:

INIT *expression*
: sets an initial value for the variable. The expression is used only the first time the value is required. If no initial value is specified, then 0 is used by default.

>= *expression*
: sets a lower bound for the variable value. The default lower bound is $-\infty$, or 0 if the BINARY option is used.

<= *expression*
: sets an upper bound for the variable value. The default upper bound is ∞, or 1 if the BINARY option is used.

INTEGER
: requests that the solver assign the variable an integer value.

BINARY
: requests that the solver assign the variable an integer value within default bounds of 0 to 1.

For example, the following statements declare a variable that has an initial value of 0.5. The variable is bounded by 0 and 1:

```
proc optmodel;
   var x init 0.5 >= 0 <= 1;
```

The values of the bounds can be determined later by using suffixed references to the variable. For example, the upper bound for variable x can be referred to as x.ub. In addition, the bounds options can be overridden by explicit assignment to the suffixed variable name. Suffixes are described further in the section "Suffixes" on page 134. Note that the bounds for integer and binary variables are rounded to integers according to the value that you specify in the PROC OPTMODEL option INTFUZZ=.

When used in an expression, an unsuffixed variable name refers to the current value of the variable. Unsuffixed variables are not allowed in the expressions for options that define variable bounds or initial values. Such expressions have values that must be fixed during execution of the solver.

Programming Statements

PROC OPTMODEL supports several programming statements. You can perform various actions with these statements, such as reading or writing data sets, setting parameter values, generating text output, or invoking a solver.

Statements are read from the input and are executed immediately when complete. Certain statements can contain one or more substatements. The execution of substatements is held until the statements that contain them are submitted. Parameter values that are used by expressions in programming statements are resolved when the statement is executed; this resolution might cause errors to be detected. For example, the use of undefined parameters is detected during resolution of the symbolic expressions from declarations.

A statement is terminated by a semicolon. The positions at which semicolons are placed are shown explicitly in the following statement syntax descriptions.

The programming statements can be grouped into the categories shown in Table 5.7.

Table 5.7 Types of Programming Statements in PROC OPTMODEL

Control	Looping	General	Input/Output	Model
DO	COFOR	Assignment	CLOSEFILE	DROP
IF	CONTINUE	CALL	CREATE DATA	EXPAND
Null (;)	DO Iterative	PERFORMANCE	FILE	FIX
QUIT	DO UNTIL	PROFILE	PRINT	RESTORE
STOP	DO WHILE	RESET OPTIONS	PUT	SOLVE
	FOR	SUBMIT	READ DATA	UNFIX
	LEAVE		SAVE MPS	USE PROBLEM
			SAVE QPS	

Assignment Statement

identifier-expression = *expression* ;

The assignment statement assigns a variable or parameter value. The type of the target *identifier-expression* must match the type of the right-hand-side expression.

For example, the following statements set the current value for variable x to 3:

```
proc optmodel;
   var x;
   x = 3;
```

NOTE: Parameters that were declared with the equal sign (=) initialization forms must not be reassigned a value with an assignment statement. If this occurs, PROC OPTMODEL reports an error.

CALL Statement

CALL *name* **(** *argument-1* [**,** *. . . argument-n*] **) ;**

The CALL statement invokes the named library subroutine. The values that are determined for each argument expression are passed to the subroutine when the subroutine is invoked. The subroutine can update the values of PROC OPTMODEL parameters and variables when an argument is an *identifier-expression* (see the section "Identifier Expressions" on page 102). For example, the following statements set the parameter array a to a random permutation of 1 to 4:

```
proc optmodel;
   number a{i in 1..4} init i;
   number seed init -1;
   call ranperm(seed, a[1], a[2], a[3], a[4]);
```

NOTE: The maximum length of the string value returned from an output argument is equal to the character length of the argument before the call. An undefined STRING parameter that is used as an output argument has a character length of 8.

For a list of CALL routines, see *SAS Functions and CALL Routines: Reference*. You can also call subroutines that are compiled by the FCMP procedure. For more information, see the section "FCMP Routines" on page 156.

CLOSEFILE Statement

CLOSEFILE *file-specifications* **;**

The CLOSEFILE statement closes files that were opened by the FILE statement. Each file is specified by a logical name, a physical filename in quotation marks, or an expression enclosed in parentheses that evaluates to a physical filename. See the section "FILE Statement" on page 71 for more information about file specifications.

The following example shows how the CLOSEFILE statement is used with a logical filename:

```
filename greet 'hello.txt';
proc optmodel;
   file greet;
   put 'Hi!';
   closefile greet;
```

Generally you must close a file with a CLOSEFILE statement before external programs can access the file. However, any open files are automatically closed when PROC OPTMODEL terminates.

COFOR Statement

COFOR { *index-set* **}** *statement* **;**

The COFOR statement executes its *statement* for each member of the specified *index-set*, similar to how the FOR statement executes. However, in a COFOR statement, PROC OPTMODEL can execute the SOLVE statement concurrently with other statements. The execution of the COFOR substatement is interleaved between loop iterations so that other iterations can be processed while an iteration waits for a SOLVE statement to complete. Multiple solvers can run concurrently. This interleaving is managed so that in many cases a FOR loop can be replaced by a COFOR loop to achieve concurrency with minimal or no other changes to the code.

The following code shows a simple example:

```
proc optmodel printlevel=0;
   var x {1..6} >= 0;

   minimize z = sum {j in 1..6} x[j];

   con a1: x[1] + x[2] + x[3]                        <= 4;
   con a2:                     x[4] + x[5] + x[6] <= 6;
   con a3: x[1] +              x[4]               >= 5;
   con a4:        x[2] +              x[5]        >= 2;
   con a5:               x[3] +              x[6] >= 3;

   cofor{i in 3..5} do;
      fix x[1]=i;
      solve;
      put i= x[1]= _solution_status_=;
   end;
```

Figure 5.7 shows the PROC OPTMODEL output. The order of the output from different iterations can vary between runs, depending on the order in which the SOLVE statements complete. A FOR statement could have been used instead of COFOR; the FOR statement would produce a consistent output order but only one solver would execute at a time. Note that because the solver execution in this example is trivial, the benefits from concurrency are limited.

Figure 5.7 A Simple COFOR Loop

```
i=4 x[1]=4 _SOLUTION_STATUS_=OPTIMAL
i=5 x[1]=5 _SOLUTION_STATUS_=INFEASIBLE
i=3 x[1]=3 _SOLUTION_STATUS_=OPTIMAL
```

A COFOR statement can contain other control and looping statements, including nested COFOR loops. The maximum number of threads that can be used is controlled by the PERFORMANCE statement and SAS options that are in effect when the outermost COFOR loop is entered, as described in the section "Threaded and Distributed Processing" on page 160. The outermost COFOR statement allocates threads for execution on the computer that is running PROC OPTMODEL. When a PERFORMANCE statement is in effect that requests distributed computing, the outermost COFOR statement also creates a distributed execution environment that has the specified number of compute nodes. Solvers within the COFOR loop can then run remotely in single-machine mode on the compute nodes (as shown in the solver output).

The COFOR statement supports simultaneous processing of several SOLVE statements. Processing proceeds through the iteration body statements as it would through a FOR loop until a SOLVE statement that uses the CLP, LP, MILP, network, NLP, or QP solver is executed. After the problem is generated, the solver starts processing in a background thread (or remote computing node in the distributed case) and the COFOR loop switches execution to another iteration of the loop, assuming enough threads and iterations are available. (Note that you need at least two threads on the computer that is running the COFOR loop to enable overlap of statement execution with solver execution.) Execution could switch to an existing iteration where the solver has completed. Alternatively, a new iteration of a COFOR loop could be started. All output from an iteration, except within a SUBMIT block, is displayed together after the iteration has completed. Output from a SUBMIT block is displayed as the block is executed.

A COFOR loop can contain PERFORMANCE statements. These statements affect SOLVE statements that are executed subsequently in the same iteration but not those in other iterations. When a COFOR statement is running in distributed mode, the default value of the NTHREADS= option in the PERFORMANCE statement is the number of threads per compute node, as determined when the outermost COFOR loop starts. Otherwise the default value of the NTHREADS= option is 1. Executing SOLVE statements in the background by using a single thread usually provides the best performance on a single computer.

Each iteration of a COFOR loop begins execution by setting default performance options. In effect, each iteration of a COFOR loop begins with an implicit PERFORMANCE statement,

```
performance parallelmode=<outer-mode> <details>;
```

where PARALLELMODE=<*outer-mode*> and <*details*> represent the PARALLELMODE= and DETAILS options, if any, in the PERFORMANCE statement in effect at the start of the COFOR loop. The NTHREADS= option is set to its default value. The following example shows how performance options are inherited:

```
performance nodes=10 nthreads=16
            parallelmode=nondeterministic;
cofor {iter in ITERSET} do;
   /* implicit statement, uses default NTHREADS=16 */
   * performance parallelmode=nondeterministic;

   /* set up problem */
   . . .

   /* runs with NTHREADS=16, nondeterministic */
   solve with lp/algorithm=con;

   /* change problem */
   . . .

   /* reset NTHREADS=, default PARALLELMODE=DETERMINISTIC */
   performance nthreads=8;

   /* runs with NTHREADS=8, deterministic */
   solve with lp/algorithm=ip;
end;

/* the PERFORMANCE statement preceding the COFOR resumes effect */
```

The order in which the solvers complete is unpredictable. So it is usually not useful for a problem that is solved within an iteration to depend on the results of SOLVE statements that are executed in other iterations of

the COFOR loop. It is advisable to limit global parameter updates to operations where order is not important, such as accumulating counts, sums, or unions or writing mutually exclusive subsets of an array. It is possible to execute multiple SOLVE statements within a loop iteration, and subsequent solver invocations within an iteration can use results from prior solvers in the same iteration.

In many cases, a COFOR loop iteration solves a specialized version of a common problem structure. This requires it to modify problem attributes that are also used in other iterations, such as coefficient values or the fixed status of variables. Changes to problem attributes are not made visible to other iterations of a COFOR loop in order to avoid confusing behavior due to interleaved execution. For example, the value printed for x[1] in Figure 5.7 is the local value for the iteration, not the most recent global value. Changes to these attributes create or update a copy of the value that is local to the iteration. These attribute values along with the local dummy parameters provide a local context for the iteration.

The following problem attributes are automatically made local to the modifying iteration when they are changed within a COFOR loop:

- the current problem, selected by USE PROBLEM
- the value of variables and their suffix values
- the fixed status of variables
- the constraint suffix values
- the dropped status of constraints
- the .LABEL suffix
- NUMBER, STRING, and SET parameters that determine values that are used in the bounds or body expressions of problem declarations (CONSTRAINT, IMPVAR, MIN, MAX, or VAR)
- NUMBER, STRING, and SET parameters that determine values that are used in solver arguments within the same outermost COFOR loop
- the predeclared string parameters _SOLVER_OPTIONS_ and _*solver*_OPTIONS_ (for each solver)

To illustrate these rules, consider the following code, which uses the NLP solver to solve a MINLP portfolio optimization problem by selecting random subsets of the assets to optimize:

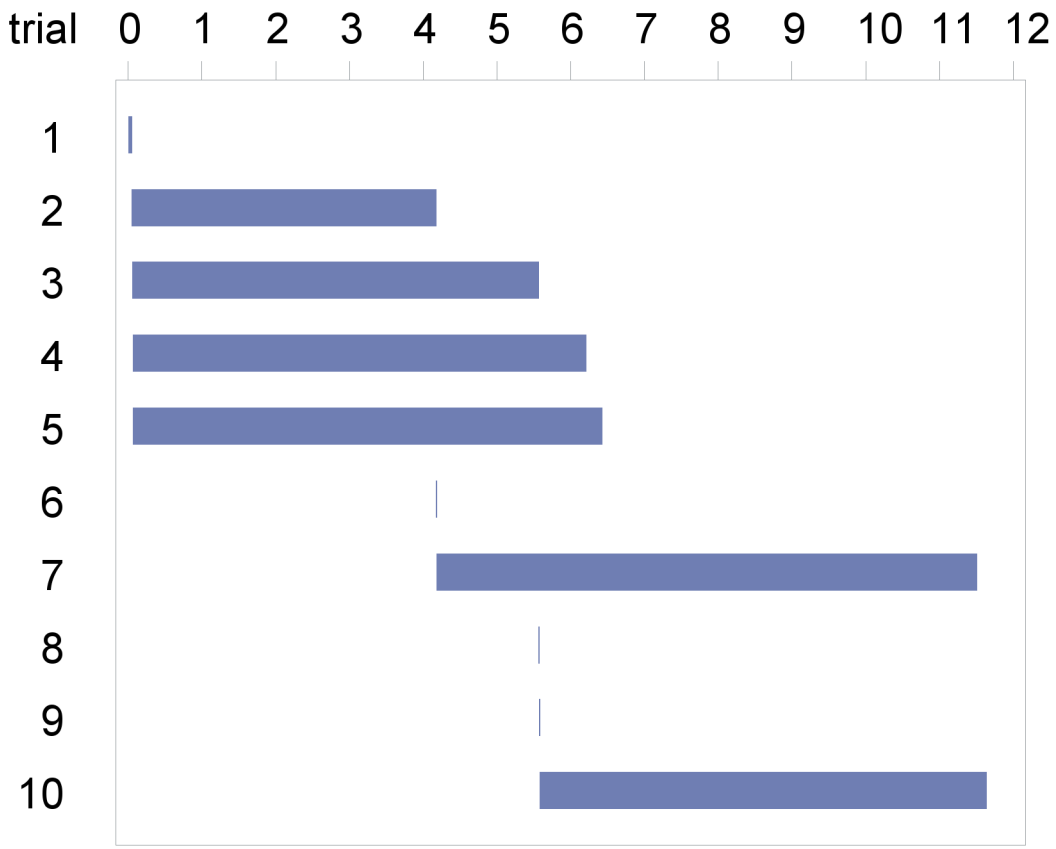
```
proc optmodel printlevel=0;
   /* assets and related parameters */
   set ASSETS;
   num return {ASSETS};
   num cov {ASSETS, ASSETS} init 0;
   read data means into ASSETS=[_n_] return;
   read data covdata into [asset1 asset2] cov cov[asset2,asset1]=cov;
   num riskLimit init 0.00025;
   num minThreshold init 0.1;
   num numTrials = 10;

   /* number of random trials */
   set TRIALS = 1..numTrials;
```

```
   /* declare NLP problem for fixed set of assets */
   set ASSETS_THIS;
   var AssetPropVar {ASSETS} >= minThreshold <= 1;
   max ExpectedReturn = sum {i in ASSETS} return[i] * AssetPropVar[i];
   con RiskBound:
      sum {i in ASSETS_THIS, j in ASSETS_THIS}
          cov[i,j] * AssetPropVar[i] * AssetPropVar[j] <= riskLimit;
   con TotalPortfolio:
      sum {asset in ASSETS} AssetPropVar[asset] = 1;

   /* parameters to track best solution */
   num infinity = constant('BIG');
   num best_objective init -infinity;
   set INCUMBENT;

   /* iterate over trials */
   num start  {TRIALS};
   num finish {TRIALS};
   num overall_start;
   overall_start = time();
   call streaminit(1);
   cofor {trial in TRIALS} do;
      start[trial] = time() - overall_start;
      put;
      put trial=;
      ASSETS_THIS = {i in ASSETS: rand('UNIFORM') < 0.5};
      put ASSETS_THIS=;
      for {i in ASSETS diff ASSETS_THIS}
         fix AssetPropVar[i] = 0;
      solve with NLP / logfreq=0;
      put _solution_status_=;
      if _solution_status_ ne 'INFEASIBLE' then do;
         if best_objective < ExpectedReturn then do;
            best_objective = ExpectedReturn;
            INCUMBENT = ASSETS_THIS;
         end;
      end;
      finish[trial] = time() - overall_start;
   end;

   put best_objective= INCUMBENT=;
   create data ganttdata from [trial] e_start=start e_finish=finish;

proc gantt data=ganttdata;
   id trial;
   chart / compress nolegend nojobnum mindate=0 top height=1.8;
run;
```

All the COFOR loop iterations use the same problem, _START_. However, the changes to the problem are local to the iteration that makes them. For example, the FIX statement does not affect variables in other iterations. The value of the ASSETS_THIS parameter is used by the RiskBound constraint, so the change to it is local. Because AssetPropVar is a VAR, the changes to its value are also local.

On the other hand, the values of the best_objective and INCUMBENT parameters do not affect any problem declarations. Therefore, their global values are used, enabling the code in the COFOR loop to select and save the best result. Similarly, the start and finish parameters are not used in the problem and allow the overlapping of iterations to be illustrated. Figure 5.8 from the GANTT procedure shows how the iterations have overlapped execution times.

Figure 5.8 Overlapped COFOR Iterations



Changes to problem attributes from completed iterations are made visible after the loop is finished. They appear in the context that contained the COFOR statement. If multiple iterations modify the same problem attribute value, then the value from the iteration that completed last is the one made visible.

The LEAVE statement can be used to terminate execution of a COFOR loop. This completes the current iteration of the COFOR loop. The currently active solvers for the COFOR loop are terminated, and the output of the incomplete iterations is discarded. The CONTINUE statement within a COFOR loop can also be used to complete the current iteration, but it has no effect on other iterations.

Using the LEAVE statement to terminate is useful, for example, when a sufficiently good solution is found for a problem. The preceding code has been modified as follows to keep generating solutions until a time limit is reached. The code sets a time limit and then executes the LEAVE statement to stop processing when the limit is exceeded. The COFOR loop uses a very large iteration range to allow it to run indefinitely.

```
proc optmodel printlevel=0;
   set ASSETS;
   num return {ASSETS};
   num cov {ASSETS, ASSETS} init 0;
   read data means into ASSETS=[_n_] return;
   read data covdata into [asset1 asset2] cov cov[asset2,asset1]=cov;
   num riskLimit init 0.00025;
   num minThreshold init 0.1;

   /* declare NLP problem for fixed set of assets */
   set ASSETS_THIS;
   var AssetPropVar {ASSETS} >= minThreshold <= 1;
   max ExpectedReturn = sum {i in ASSETS} return[i] * AssetPropVar[i];
   con RiskBound:
      sum {i in ASSETS_THIS, j in ASSETS_THIS}
          cov[i,j] * AssetPropVar[i] * AssetPropVar[j] <= riskLimit;
   con TotalPortfolio:
      sum {asset in ASSETS} AssetPropVar[asset] = 1;

   num infinity = constant('BIG');
   num best_objective init -infinity;
   set INCUMBENT;

   /* run for 30 seconds */
   num last_time;
   last_time = time() + 30;
   num n_trials init 0;
   call streaminit(1);
   cofor {trial in 1..1e9} do;
      put;
      put trial=;
      ASSETS_THIS = {i in ASSETS: rand('UNIFORM') < 0.5};
      put ASSETS_THIS=;
      for {i in ASSETS diff ASSETS_THIS} fix AssetPropVar[i] = 0;
      solve with NLP / logfreq=0;
      put _solution_status_=;
      if _solution_status_ ne 'INFEASIBLE' then do;
         if best_objective < ExpectedReturn then do;
            best_objective = ExpectedReturn;
            INCUMBENT = ASSETS_THIS;
         end;
      end;
      n_trials = n_trials + 1;
      if time() >= last_time then leave;
   end;

   put n_trials=;
   put best_objective= INCUMBENT=;
quit;
```

CONTINUE Statement

CONTINUE ;

The CONTINUE statement terminates the current iteration of the loop statement (iterative DO, DO UNTIL, DO WHILE, FOR, or COFOR) that immediately contains the CONTINUE statement. Execution resumes at the start of the loop after checking WHILE or UNTIL tests. The FOR, COFOR, or iterative DO loops apply new iteration values.

CREATE DATA Statement

CREATE DATA *SAS-data-set* **FROM** [**[** *key-columns* **]** [**=** *key-set*]] *columns* **;**

The CREATE DATA statement creates a new SAS data set and copies data into it from PROC OPTMODEL parameters and variables. The CREATE DATA statement can create a data set with a single observation or a data set with observations for every location in one or more arrays. The data set is closed after the execution of the CREATE DATA statement.

The arguments to the CREATE DATA statement are as follows:

SAS-data-set
: specifies the output data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses.

key-columns
: declares index values and their corresponding data set variables. The values are used to index array locations in *columns*.

key-set
: specifies a set of index values for the *key-columns*.

columns
: specifies data set variables as well as the PROC OPTMODEL source data for the variables.

Each *column* or *key-column* defines output data set variables and a data source for a column. For example, the following statement generates the output SAS data set resdata from the PROC OPTMODEL array opt, which is indexed by the set indset:

```
create data resdata from [solns]=indset opt;
```

The output data set variable solns contains the index elements in indset.

Columns

Columns can have the following forms:

identifier-expression [/ *options*]
: transfers data from the PROC OPTMODEL parameter or variable specified by the *identifier-expression*. The output data set variable has the same name as the *name* part of the *identifier-expression* (see the section "Identifier Expressions" on page 102). If the *identifier-expression* refers to an array, then the index can be omitted when it matches the *key-columns*. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 63 for more information. The following example creates a data set with the variables m and n:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from m n;
```

name **=** *expression* [/ *options*]

transfers the value of a PROC OPTMODEL expression to the output data set variable *name*. The *expression* is reevaluated for each observation. If the *expression* contains any operators or function calls, then it must be enclosed in parentheses. If the *expression* is an *identifier-expression* that refers to an array, then the index can be omitted if it matches the *key-columns*. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 63 for more information. The following example creates a data set with the variable ratio:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from ratio=(m/n);
```

COL(*name-expression***) =** *expression* [/ *options*]

transfers the value of a PROC OPTMODEL expression to the output data set variable named by the string expression *name-expression*. The PROC OPTMODEL expression is reevaluated for each observation. If this expression contains any operators or function calls, then it must be enclosed in parentheses. If the PROC OPTMODEL *expression* is an *identifier-expression* that refers to an array, then the index can be omitted if it matches the *key-columns*. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 63 for more information. The following example uses the COL expression to form the variable s5:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from col("s"||n)=(m+n);
```

{ *index-set* **} <** *columns* **>**

performs the transfers by iterating each column specified by **<** *columns* **>** for each member of the *index set*. If there are n columns and m index set members, then $n \times m$ columns are generated. The dummy parameters from the index set can be used in the columns to generate distinct output data set variable names in the iterated columns, using COL expressions. The columns are expanded when the CREATE DATA statement is executed, before any output is performed. This form of *columns* cannot be nested. In other words, the following form of *columns* is NOT allowed:

{ *index-set* **} <{** *index-set* **} <** *columns* **> >**

The following example demonstrates the use of the iterated *columns* form:

```
proc optmodel;
   set<string> alph = {'a', 'b', 'c'};
   var x{1..3, alph} init 2;
   create data example from [i]=(1..3)
      {j in alph}<col("x"||j)=x[i,j]>;
```

The data set created by these statements is shown in Figure 5.9.

Figure 5.9 CREATE DATA with COL Expression

Obs	i	xa	xb	xc
1	1	2	2	2
2	2	2	2	2
3	3	2	2	2

NOTE: When no *key-columns* are specified, the output data set has a single observation.

The following statements incorporate several of the preceding examples to create and print a data set by using PROC OPTMODEL parameters:

```
proc optmodel;
   number m = 7, n = 5;
   create data example from m n ratio=(m/n) col("s"||n)=(m+n);

proc print;
run;
```

The output from the PRINT procedure is shown in Figure 5.10.

Figure 5.10 CREATE DATA for Single Observation

Obs	m	n	ratio	s5
1	7	5	1.4	12

Column Options

Each *column* or *key-column* that defines a data set variable can be followed by zero or more of the following modifiers:

FORMAT=*format.*
: associates a format with the current column.

INFORMAT=*informat.*
: associates an informat with the current column.

LABEL=*'label'*
: associates a label with the current column. The label can be specified by a quoted string or an expression in parentheses.

LENGTH=*length*
: specifies a length for the current column. The length can be specified by a numeric constant or a parenthesized expression. The range for character variables is 1 to 32,767 bytes. The range for numeric variables depends on the operating environment and has a minimum of 2 or 3.

TRANSCODE=YES | **NO**
: specifies whether character variables can be transcoded. The default value is YES. See the TRANSCODE=option of the ATTRIB statement in *SAS Statements: Reference* for more information.

The following statements demonstrate the use of column options, including the use of multiple options for a single column:

```
proc optmodel;
   num sq{i in 1..10} = i*i;
   create data squares from [i/format=hex2./length=3] sq/format=6.2;

proc print;
run;
```

The output from the PRINT procedure is shown in Figure 5.11.

Figure 5.11 CREATE DATA with Column Options

Obs	i	sq
1	01	1.00
2	02	4.00
3	03	9.00
4	04	16.00
5	05	25.00
6	06	36.00
7	07	49.00
8	08	64.00
9	09	81.00
10	0A	100.00

Key Columns

Key-columns declare index values that enable multiple observations to be written from array *columns*. An observation is created for each unique index value combination. The index values supply the index for array *columns* that do not have an explicit index.

Key-columns define the data set variables where the index value elements are written. They can also declare local dummy parameters for use in expressions in the *columns*. *Key-columns* are syntactically similar to *columns*, but are more restricted in form. The following forms of *key-columns* are allowed:

name [/ *options*]
: transfers an index element value to the data set variable *name*. A local dummy parameter, *name*, is declared to hold the index element value. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 63 for more information.

COL(*name-expression***)** [**=** *index-name*] [/ *options*]
: transfers an index element value to the data set variable named by the string-valued *name-expression*. The argument *index-name* optionally declares a local dummy parameter to hold the index element value. The *options* enable formats and labels to be associated with the data set variable. See the section "Column Options" on page 63 for more information.

A *key-set* in the CREATE DATA statement explicitly specifies the set of index values. *key-set* can be specified as a set expression, although it must be enclosed in parentheses if it contains any function calls or operators. *key-set* can also be specified as an index set expression, in which case the *index-set* dummy parameters override any dummy parameters that are declared in the *key-columns* items. The following statements create a data set from the PROC OPTMODEL parameter m, a matrix whose only nonzero entries are located at (1, 1) and (4, 1):

```
proc optmodel;
   number m{1..5, 1..3} = [[1 1] 1 [4 1] 1];
   create data example
      from [i j] = {setof{i in 1..2}<i**2>, {1, 2}} m;

proc print data=example noobs;
run;
```

The dummy parameter i in the SETOF expression takes precedence over the dummy parameter i declared in the *key-columns* item. The output from these statements is shown in Figure 5.12.

Figure 5.12 CREATE: *key-set* with SETOF Aggregation Expression

i	j	m
1	1	1
1	2	0
4	1	1
4	2	0

If no *key-set* is specified, then the set of index values is formed from the union of the index sets of the implicitly indexed *columns*. The number of index elements for each implicitly indexed array must match the number of *key-columns*. The type of each index element (string versus numeric) must match the element of the same position in other implicit indices.

The arrays for implicitly indexed columns in a CREATE DATA statement do not need to have identical index sets. A missing value is supplied for the value of an implicitly indexed array location when the implied index value is not in the array's index set.

In the following statements, the *key-set* is unspecified. The set of index values is $\{1, 2, 3\}$, which is the union of the index sets of x and y. These index sets are not identical, so missing values are supplied when necessary. The results of these statements are shown in Figure 5.13.

```
proc optmodel;
   number x{1..2} init 2;
   var y{2..3} init 3;
   create data exdata from [keycol] x y;

proc print;
run;
```

Figure 5.13 CREATE: Unspecified *key-set*

Obs	keycol	x	y
1	1	2	.
2	2	2	3
3	3	.	3

The types of the output data set variables match the types of the source values. The output variable type for a *key-columns* matches the corresponding element type in the index value tuple. A numeric element matches a NUMERIC data set variable, while a string element matches a CHAR variable. For regular *columns* the source expression type determines the output data set variable type. A numeric expression produces a NUMERIC variable, while a string expression produces a CHAR variable.

Lengths of character variables in the output data set are determined automatically. The length is set to accommodate the longest string value output in that column.

You can use the iterated *columns* form to output selected rows of multiple arrays, assigning a different data set variable to each column. For example, the following statements output the last two rows of the two-dimensional array, a, along with corresponding elements of the one-dimensional array, b:

```
proc optmodel;
   num m = 3;  /* number of rows/observations */
   num n = 4;  /* number of columns in a */
   num a{i in 1..m, j in 1..n} = i*j;  /* compute a */
   num b{i in 1..m} = i**2;  /* compute b */
   set<num> subset = 2..m;  /* used to omit first row */
   create data out
          from [i]=subset {j in 1..n}<col("a"||j)=a[i,j]> b;
```

The preceding statements create a data set out, which has $m - 1$ observations and $n + 2$ variables. The variables are named i, a1 through an, and b, as shown in Figure 5.14.

Figure 5.14 CREATE DATA Set: The Iterated Column Form

Obs	i	a1	a2	a3	a4	b
1	2	2	4	6	8	4
2	3	3	6	9	12	9

See the section "Data Set Input/Output" on page 117 for more examples of using the CREATE DATA statement.

DO Statement

DO ; *statements* **; END ;**

The DO statement groups a sequence of statements together as a single statement. Each statement within the list is executed sequentially. The DO statement can be used for grouping with the IF, FOR, and COFOR statements.

DO Statement, Iterative

DO *name* **=** *specification-1* [**,** *. . . specification-n*] **;** *statements* **; END ;**

The iterative DO statement assigns the values from the sequence of *specification* items to a previously declared parameter or variable, *name*. The specified statement sequence is executed after each assignment. This statement corresponds to the iterative DO statement of the DATA step.

Each *specification* provides either a single number or a single string value, or a sequence of such values. Each *specification* takes the following form:

expression [**WHILE(** *logic-expression* **)** | **UNTIL(** *logic-expression* **)**]

The *expression* in the *specification* provides a single value or set of values to assign to the target *name*. Multiple values can be provided for the loop by giving multiple *specification* items that are separated by commas. For example, the following statements output the values 1, 3, and 5:

```
proc optmodel;
   number i;
   do i=1,3,5;
      put i;
   end;
```

In this case, the same effect can be achieved with a single range expression in place of the explicit list of values, as in the following statements:

```
proc optmodel;
   number i;
   do i=1 to 5 by 2;
      put 'value of i assigned by the DO loop = ' i;
      i=i**2;
      put 'value of i assigned in the body of the loop = ' i;
   end;
```

The output of these statements is shown in Figure 5.15.

Figure 5.15 DO Loop: Name Parameter Unaffected

```
value of i assigned by the DO loop = 1
value of i assigned in the body of the loop = 1
value of i assigned by the DO loop = 3
value of i assigned in the body of the loop = 9
value of i assigned by the DO loop = 5
value of i assigned in the body of the loop = 25
```

Unlike the DATA step, a range expression requires the limit to be specified. Additionally the BY part, if any, must follow the limit expression. Moreover, although the *name* parameter can be reassigned in the body of the loop, the sequence of values that is assigned by the DO loop is unaffected.

The argument *expression* can also be an expression that returns a set of numbers or strings. For example, the following statements produce the same sequence of values for i as the previous statements but use a set parameter value:

```
proc optmodel;
   set s = {1,3,5};
   number i;
   do i = s;
      put i;
   end;
```

Each *specification* can include a WHILE or UNTIL clause. A WHILE or UNTIL clause applies to the *expression* that immediately precedes the clause. The sequence that is specified by an *expression* can be terminated early by a WHILE or UNTIL clause. A WHILE *logic-expression* is evaluated for each sequence value before the nested *statements*. If the *logic-expression* returns a false (zero or missing) value, then the current sequence is terminated immediately. An UNTIL *logic-expression* is evaluated for each sequence value after the nested *statements*. The sequence from the current *specification* is terminated if the *logic-expression* returns a true value (nonzero and nonmissing). After early termination of a sequence due to a WHILE or UNTIL expression, the DO loop execution continues with the next *specification*, if any.

To demonstrate use of the WHILE clause, the following statements output the values 1, 2, and 3. In this case the sequence of values from the set s is stopped when the value of i reaches 4.

```
proc optmodel;
   set s = {1,2,3,4,5};
   number i;
   do i = s while(i NE 4);
      put i;
   end;
```

DO UNTIL Statement

DO UNTIL (*logic-expression* **) ;** *statements* **; END ;**

The DO UNTIL loop executes the specified sequence of statements repeatedly until the *logic-expression*, evaluated after the *statements*, returns true (a nonmissing nonzero value).

For example, the following statements output the values 1 and 2:

```
proc optmodel;
   number i;
   i = 1;
   do until (i=3);
      put i;
      i=i+1;
   end;
```

Multiple criteria can be introduced using expression operators, as in the following example:

```
do until (i=3 and j=7);
```

For a list of expression operators, see Table 5.10.

DO WHILE Statement

DO WHILE (*logic-expression* **) ;** *statements* **; END ;**

The DO WHILE loop executes the specified sequence of statements repeatedly as long as the *logic-expression*, evaluated before the *statements*, returns true (a nonmissing nonzero value).

For example, the following statements output the values 1 and 2:

```
proc optmodel;
   number i;
   i = 1;
   do while (i<3);
      put i;
      i=i+1;
   end;
```

Multiple criteria can be introduced using expression operators, as in the following example:

```
do while (i<3 and j<7);
```

For a list of expression operators, see Table 5.10.

DROP Statement

DROP *identifier-list* **;**

The DROP statement causes the solver to ignore a list of constraints, constraint arrays, or constraint array locations. The space-delimited *identifier-list* specifies the names of the dropped constraints. Each constraint, constraint array, or constraint array location is named by an *identifier-expression*. An entire constraint array is dropped if an *identifier-expression* omits the index for an array name.

The following example statements use the DROP statement:

```
proc optmodel;
   var x{1..10};
   con c1: x[1] + x[2] <= 3;
   con disp{i in 1..9}: x[i+1] >= x[i] + 0.1;

   drop c1;         /* drops the c1 constraint */
   drop disp[5];    /* drops just disp[5] */
   drop disp;       /* drops all disp constraints */
```

The constraint can be added back to the model with the RESTORE statement.

The following line drops both the c1 and disp[5] constraints:

```
drop c1 disp[5];
```

EXPAND Statement

EXPAND [*identifier-expression*] [/ *options*] **;**

The EXPAND statement prints the specified constraint, variable, implicit variable, or objective declaration expressions in the current problem after expanding aggregation operators, substituting the current value for parameters and indices, and resolving constant subexpressions. *identifier-expression* is the name of a variable, objective, or constraint. If the name is omitted and no *options* are specified, then all variables, objectives, implicit variables, and undropped constraints in the current problem are printed. The following statements show an example EXPAND statement:

```
proc optmodel;
   number n=2;
   var x{1..n};
   min z1=sum{i in 1..n}(x[i]-i)**2;
   max z2=sum{i in 1..n}(i-x[i])**3;
   con c{i in 1..n}: x[i]>=0;
   fix x[2]=3;
   expand;
```

These statements produce the output in Figure 5.16.

Figure 5.16 EXPAND Statement Output

```
Var x[1]
Fix x[2] = 3
Maximize z2=(-x[1] + 1)**3 + (-x[2] + 2)**3
Constraint c[1]: x[1] >= 0
Constraint c[2]: x[2] >= 0
```

Specifying an *identifier-expression* restricts output to the specified declaration. A non-array name prints only the specified item. If an array name is used with a specific index, then information for the specified array location is output. Using an array name without an index restricts output to all locations in the array.

You can use the following *options* to further control the EXPAND statement output:

SOLVE
causes the EXPAND statement to print the variables, objectives, and constraints in the same form that would be seen by the solver if a SOLVE statement were executed. This includes any transformations by the PROC OPTMODEL presolver (see the section "Presolver" on page 145). In this form any fixed variables are replaced by their values. Unless an *identifier-expression* specifies a particular non-array item or array location, the EXPAND output is restricted to only the variables, the constraints, and the current problem objective.

The following options restrict the types of declarations output when no specific non-array item or array location is requested. By default, all types of declarations are output. Only the requested declaration types are output when one or more of the following options are used.

CONSTRAINT | CON
requests the output of undropped constraints.

FIX
requests the output of fixed variables. These variables might have been fixed by the FIX statement (or by the presolver if the SOLVE option is specified). The FIX option can also be used in combination with the name of a variable array to display just the fixed elements of the array.

IIS
restricts the display to items found in the irreducible infeasible set (IIS) after the most recent SOLVE performed by the LP solver with the IIS=ON option. The IIS option for the EXPAND statement can also be used in combination with the name of a variable or constraint array to display only the elements of the array in the IIS. For more information about IIS, see the section "Irreducible Infeasible Set" on page 274.

IMPVAR
requests the output of implicit variables referenced in the current problem.

OBJECTIVE | OBJ
requests the output of objectives used in the current problem. This includes the current problem objective and any objectives referenced as implicit variables.

OMITTED
requests the output of variables that are referenced by problem equations but were not included in the current USE PROBLEM instance. The OPTMODEL procedure omits these variables from the generated problem.

VAR
requests the output of unfixed variables. The VAR option can also be used in combination with the name of a variable array to display just the unfixed elements of the array.

For example, you can see the effect of a FIX statement on the problem that is presented to the solver by using the SOLVE option. You can modify the previous example as follows:

```
proc optmodel;
   number n=2;
   var x{1..n};
   min z1=sum{i in 1..n}(x[i]-i)**2;
   max z2=sum{i in 1..n}(i-x[i])**3;
   con c{i in 1..n}: x[i]>=0;
   fix x[2]=3;
   expand / solve;
```

These statements produce the output in Figure 5.17.

Figure 5.17 Expansion with Fixed Variable

```
Var x[1] >= 0
Fix x[2] = 3
Maximize z2=(-x[1] + 1)**3 - 1
```

Compare the results in Figure 5.17 to those in Figure 5.16. The constraint c[1] has been converted to a variable bound. The subexpression that uses the fixed variable has been resolved to a constant.

FILE Statement

FILE *file-specification* [**LRECL=***value*] **;**

The FILE statement selects the current output file for the PUT statement. By default PUT output is sent to the SAS log. Use the FILE statement to manage a group of output files. The specified file is opened for output if it is not already open. The output file remains open until it is closed with the CLOSEFILE statement.

file-specification names the output file. It can use any of the following forms:

'*external-file*'
: specifies the physical name of an external file in quotation marks. The interpretation of the filename depends on the operating environment.

file-name
: specifies the logical name associated with a file by the FILENAME statement or by the operating environment. The names PRINT and LOG are reserved to refer to the SAS listing and log files, respectively.

  NOTE: Details about the FILENAME statement can be found in *SAS Statements: Reference*.

(*expression* **)**
: specifies an expression that evaluates to a string that contains the physical name of an external file.

The LRECL= option sets the line length of the output file. The LRECL= option is ignored if the file is already open or if the PRINT or LOG file is specified.

The LRECL= *value* can be specified in these forms:

integer
: specifies the desired line length.

identifier-expression
: specifies the name of a numeric parameter that contains the length.

(*expression* **)**
: specifies a numeric expression in parentheses that returns the line length.

The LRECL= *value* cannot exceed the largest four-byte signed integer, which is $2^{31} - 1$.

The following example shows how to use the FILE statement to handle multiple files:

```
proc optmodel;
   file 'file.txt' lrecl=80;   /* opens file.txt     */
   put 'This is line 1 of file.txt.';
   file print;                  /* selects the listing */
   put 'This goes to the listing.';
   file 'file.txt';            /* reselects file.txt */
   put 'This is line 2 of file.txt.';
   closefile 'file.txt';       /* closes file.txt    */
   file log;                    /* selects the SAS log */
   put 'This goes to the log.';

   /* using expression to open and write a collection of files */
   str ofile;
   num i;
   num l = 40;
   do i = 1 to 3;
      ofile = ('file' || i || '.txt');
      file (ofile) lrecl=(l*i);
      put ('This goes to ' || ofile);
      closefile (ofile);
   end;
```

The following statements illustrate the usefulness of using a logical name associated with a file by FILENAME statement:

```
proc optmodel;
   /* assigns a logical name to file.txt */
   /* see FILENAME statement in */
   /* SAS Statements: Reference */
   filename myfile 'file.txt' mod;

   file myfile;
   put 'This is line 3 of file.txt.';
   closefile myfile;
   file myfile;
   put 'This is line 4 of file.txt.';
   closefile myfile;
```

Notice that the FILENAME statement opens the file referenced for append. Therefore, new data are appended to the end every time the logical name, myfile, is used in the FILE statement.

FIX Statement

FIX *identifier-list* [**= (** *expression* **)**] **;**

The FIX statement causes the solver to treat a list of variables, variable arrays, or variable array locations as fixed in value. The *identifier-list* consists of one or more variable names separated by spaces. Each member of the *identifier-list* is fixed to the same *expression*. For example, the following statements fix the variables x and y to 3:

```
proc optmodel;
   var x, y;
   num a = 2;
   fix x y=(a+1);
```

A variable is specified with an *identifier-expression* (see the section "Identifier Expressions" on page 102). An entire variable array is fixed if the *identifier-expression* names an array without providing an index. A new value can be specified with the *expression*. If the *expression* is a constant, then the parentheses can be omitted. For example, the following statements fix all locations in array x to 0 except x[10], which is fixed to 1:

```
proc optmodel;
   var x{1..10};
   fix x = 0;
   fix x[10] = 1;
```

If *expression* is omitted, the variable is fixed at its current value. For example, you can fix some variables to be their optimal values after the SOLVE statement is invoked. **NOTE:** The fixed value is equal to the current value for a fixed variable. The fixed value is updated if a new value is assigned to a fixed variable.

The effect of FIX can be reversed by using the UNFIX statement.

FOR Statement

FOR { *index-set* **}** *statement* **;**

The FOR statement executes its substatement for each member of the specified *index-set*. The index set can declare local dummy parameters. You can reference the value of these parameters in the substatement. For example, consider the following statements:

```
proc optmodel;
    for {i in 1..2, j in {'a', 'b'}} put i= j=;
```

These statements produce the output in Figure 5.18.

Figure 5.18 FOR Statement Output

```
i=1 j=a
i=1 j=b
i=2 j=a
i=2 j=b
```

As another example, the following statements set the current values for variable x to random values between 0 and 1:

```
proc optmodel;
   var x{1..10};
   for {i in 1..10}
       x[i] = ranuni(-1);
```

Multiple statements can be controlled by specifying a DO statement group for the substatement.

CAUTION: Avoid modifying the parameters that are used by the FOR or COFOR statement index set from within the substatement. The set value that is used for the left-most index set item is not affected by such changes. However, the effect of parameter changes on later index set items cannot be predicted.

IF Statement

IF *logic-expression* **THEN** *statement* [**ELSE** *statement*] **;**

The IF statement evaluates the logical expression and then conditionally executes the THEN or ELSE substatements. The substatement that follows the THEN keyword is executed when the logical expression result is nonmissing and nonzero. The ELSE substatement, if any, is executed when the logical expression result is a missing value or zero. The ELSE part is optional and must immediately follow the THEN substatement. When IF statements are nested, an ELSE is always matched to the nearest incomplete unmatched IF-THEN. Multiple statements can be controlled by using DO statements with the THEN or ELSE substatements.

NOTE: When an IF-THEN statement is used **without** an ELSE substatement, substatements of the IF statement are executed when possible as they are entered. Under certain circumstances, such as when an IF statement is nested in a FOR loop, the statement is not executed during interactive input until the next statement is seen. By following the IF-THEN statement with an extra semicolon, you can cause it to be executed upon submission, since the extra semicolon is handled as a null statement.

LEAVE Statement

LEAVE ;

The LEAVE statement terminates the execution of the entire loop body (iterative DO, DO UNTIL, DO WHILE, FOR, or COFOR) that immediately contains the LEAVE statement. Execution resumes at the statement that follows the loop. The following example demonstrates a simple use of the LEAVE statement:

```
proc optmodel;
   number i, j;
   do i = 1..5;
      do j = 1..4;
         if i >= 3 and j = 2 then leave;
      end;
   print i j;
   end;
```

The results from these statements are displayed in Figure 5.19.

Figure 5.19 LEAVE Statement Output

i	j
1	4

Figure 5.19 *continued*

i	j
2	4

i	j
3	2

i	j
4	2

i	j
5	2

For values of i equal to 1 or 2, the inner loop continues uninterrupted, leaving j with a value of 4. For values of i equal to 3, 4, or 5, the inner loop terminates early, leaving j with a value of 2.

Null Statement

;

The null statement is treated as a statement in the PROC OPTMODEL syntax, but its execution has no effect. It can be used as a placeholder statement.

PERFORMANCE Statement

PERFORMANCE *options* **;**

The PERFORMANCE statement controls the multithreaded and distributed execution features of PROC OPTMODEL and its solvers. The *options* that you specify in the PERFORMANCE statement are applied each time the statement is executed; they replace any previously specified options. For details about the options available for the PERFORMANCE statement, see the section "PERFORMANCE Statement" on page 21.

Within a COFOR loop, the PERFORMANCE statement controls solvers that are executed subsequently in the same COFOR iteration. The PERFORMANCE statement cannot specify distributed computing options in this context. If the COFOR loop is running in distributed mode, the options affect the solver as it runs in single-machine mode on the remote computing node. In distributed mode, the default NTHREADS= option value is equal to the number of threads on a node in the distributed computing environment. If the COFOR loop is running in multithreaded mode, the options affect execution of the solver on the machine that is running PROC OPTMODEL. In multithreaded mode, the default NTHREADS= option value is 1. In either mode, the NTHREADS= option value is limited to the number of threads in the COFOR loop execution environment.

PRINT Statement

PRINT *print-items* **;**

The PRINT statement outputs string and numeric data in tabular form. The statement specifies a list of arrays or other data items to print. Multiple items can be output together as data columns in the same table.

If no format is specified, the PRINT statement handles the details of formatting automatically (see the section "Formatted Output" on page 121 for details). The default format for a numerical column is the fixed-point format (*w.d* format), which is chosen based on the values of the PDIGITS= and PWIDTH= options (see the section "PROC OPTMODEL Statement" on page 40) and on the values in the column. The PRINT statement uses scientific notation (the E*w.* format) when a value is too large or too small to display in fixed format. The default format for a character column is the $*w.* format, where the width is set to be the length of the longest string (ignoring trailing blanks) in the column.

print-item can be specified in the following forms:

identifier-expression [*format*]
: specifies a data item to output. *identifier-expression* can name an array. In that case all defined array locations are output. *format* specifies a SAS format that overrides the default format.

(*expression* **)** [*format*]
: specifies a data value to output. *format* specifies a SAS format that overrides the default format.

{ *index-set* **}** *identifier-expression* [*format*]
: specifies a data item to output under the control of an *index set*. The item is printed as if it were an array with the specified set of indices. This form can be used to print a subset of the locations in an array, such as a single column. If the *identifier-expression* names an array, then the indices of the array must match the indices of the *index-set*. The *format* argument specifies a SAS format that overrides the default format.

{ *index-set* **} (** *expression* **)** [*format*]
: specifies a data item to output under the control of an *index set*. The item is printed as if it were an array with the specified set of indices. In this form the *expression* is evaluated for each member of the *index-set* to create the array values for output. *format* specifies a SAS format that overrides the default format.

string
: specifies a string value to print.

PAGE
: specifies a page break.

The following example demonstrates the use of several *print-item* forms:

```
proc optmodel;
   num x = 4.3;
   var y{j in 1..4} init j*3.68;
   print y; /* identifier-expression */
   print (x * .265) dollar6.2; /* (expression) [format] */
   print {i in 2..4} y; /* {index-set} identifier-expression */
   print {i in 1..3}(i + i*.2345692) best7.;
                              /* {index-set} (expression) [format] */
   print "Line 1"; /* string */
```

The output is displayed in Figure 5.20.

Figure 5.20 *Print-item* Forms

[1]	y
1	3.68
2	7.36
3	11.04
4	14.72

$1.14

[1]	y
2	7.36
3	11.04
4	14.72

[1]	
1	1.23457
2	2.46914
3	3.70371

Line 1

Adjacent print items that have similar indexing are grouped together and output in the same table. Items have similar indexing if they specify arrays that have the same number of indices and have matching index types (numeric versus string). Nonarray items are considered to have the same indexing as other nonarray items. The resulting table has a column for each array index followed by a column for each print item value. This format is called *list form*. For example, the following statements produce a list form table:

```
proc optmodel;
   num a{i in 1..3} = i*i;
   num b{i in 3..5} = 4*i;
   print a b;
```

These statements produce the listing output in Figure 5.21.

Figure 5.21 List Form PRINT Table

[1]	a	b
1	1	
2	4	
3	9	12
4		16
5		20

The array index columns show the set of valid index values for the print items in the table. The array index column for the *i*th index is labeled [*i*]. There is a row for each combination of index values that was used. The index values are displayed in sorted ascending order.

The data columns show the array values that correspond to the index values in each row. If a particular array index is invalid or the array location is undefined, then the corresponding table entry is displayed as blank

for numeric arrays and as an empty string for string arrays. If the print items are scalar, then the table has a single row and no array index columns.

If a table contains a single array print item, the array is two-dimensional (has two indices), and the array is dense enough, then the array is shown in *matrix form*. In this format there is a single index column that contains the row index values. The label of this column is blank. This column is followed by a column for every unique column index value for the array. The latter columns are labeled by the column value. These columns contain the array values for that particular array column. Table entries that correspond to array locations that have invalid or undefined combinations of row and column indices are blank or (for strings) printed as an empty string.

The following statements generate a simple example of matrix output:

```
proc optmodel;
   print {i in 1..6, j in i..6} (i*10+j);
```

The PRINT statement produces the output in Figure 5.22.

Figure 5.22 Matrix Form PRINT Table

	1	2	3	4	5	6
1	11	12	13	14	15	16
2		22	23	24	25	26
3			33	34	35	36
4				44	45	46
5					55	56
6						66

The PRINT statement prints single two-dimensional arrays in the form that uses fewer table cells (headings are ignored). Sparse arrays are normally printed in list form, and dense arrays are normally printed in matrix form. In a PROC OPTMODEL statement, the PMATRIX= option enables you to tune how the PRINT statement displays a two-dimensional array. The value of this option scales the total number of nonempty array elements, which is used to compute the tables cells needed for list form display. Specifying values for the PMATRIX= option less than 1 causes the list form to be used in more cases, while specifying values greater than 1 causes the matrix form to be used in more cases. If the value is 0, then the list form is always used. The default value of the PMATRIX= option is 1. Changing the default can be done with the RESET OPTIONS statement.

The following statements illustrate how the PMATRIX= option affects the display of the PRINT statement:

```
proc optmodel;
   num a{i in 1..6, i..i} = i;
   num b{i in 1..3, j in 1..3} = i*j;
   print a;
   print b;
   reset options pmatrix=3;
   print a;
   reset options pmatrix=0.5;
   print b;
```

The output is shown in Figure 5.23.

Figure 5.23 PRINT Statement: Effects of PMATRIX= Option

[1]	[2]	a
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6

	b		
	1	2	3
1	1	2	3
2	2	4	6
3	3	6	9

	a					
	1	2	3	4	5	6
1	1					
2		2				
3			3			
4				4		
5					5	
6						6

[1]	[2]	b
1	1	1
1	2	2
1	3	3
2	1	2
2	2	4
2	3	6
3	1	3
3	2	6
3	3	9

From Figure 5.23, you can see that, by default, the PRINT statement tries to make the display compact. However, you can change the default by using the PMATRIX= option.

PROFILE Statement

PROFILE [*mode*] *options* **;**

The PROFILE statement controls the PROC OPTMODEL profiler, which enables you to collect and display timing and execution count information for PROC OPTMODEL processing. The profiler can be very useful for finding bottlenecks during execution, such as constraints that require large amounts of time during problem generation. When the profiler is enabled, PROC OPTMODEL records the time for processing a declaration, the time for executing statements, and the number of times that statements are executed. See "Example 5.7: Sparse Modeling" on page 182 for an example use of the PROFILE statement.

The *mode* argument specifies the action that the PROFILE statement performs. You can use the following values for *mode*:

ON
: enables the profiler. Data are collected until the profiler is disabled or PROC OPTMODEL terminates. The profiler is also enabled when no *mode* is specified in a PROFILE statement.

OFF
: disables the profiler. Note that the profiler is disabled when PROC OPTMODEL begins execution.

PRINT
: prints the current accumulated profiler data. Items for declarations and statements are displayed in a table in descending order of their net time. Accumulated data are printed automatically when PROC OPTMODEL terminates.

The *options* control how PROC OPTMODEL collects and displays profiler information. Here are the valid options:

PERCENT=*number*
: restricts the output for PROFILE PRINT to items whose net times account for at least the specified percentage of total profiled time, $\text{total time} \cdot \text{number}/100$. Items that have smaller times are aggregated into a single item at the end of the table. You can set this option before the display of profile data, and it does not affect the data collection. The value of *number* can range from 0 to 100. The default value is 1.

RESET
: discards accumulated profiler data when the PROFILE statement completes execution. Accumulated data are retained until they are explicitly reset.

STMTDEPTH=*number* | **ALL**
: allows collection of profiler data for nested statements. With the default option, STMTDEPTH=1, profiler data are collected only for top-level statements. The elapsed time for nested statement timing is included in the top-level statement timing. For example, time for a top-level FOR statement would include the execution of its substatement. Use the STMTDEPTH= option to profile the nested statements individually. The value *number* specifies the maximum nesting depth at which to profile statements individually. The nesting depth of a top-level statement is 1. Otherwise the nesting depth of a statement is one more than the nesting depth of the statement that encloses it, such as a DO, IF, or FOR statement.

  For a PROFILE statement within a DO block or DO loop, the statement depth value is interpreted relative to the enclosing DO statement. For example, specifying STMTDEPTH=1 within a DO block causes the top-level statements of the DO block to be profiled. The STMTDEPTH= option is reset to its previous value when the enclosing DO statement completes execution.

  The value of *number* can be an integer between 1 and 32,767. Using the ALL keyword is equivalent to specifying 32,767. Note that profiler timing can add significant overhead. Use a small *number* to minimize overhead.

The elapsed time that is required to process an item includes the processing of other profiled items that it depends on. For example, the processing of a constraint during problem generation might require the

evaluation of parameter values. The PROFILE statement reports net time so that the total of profiled times represents actual processing time.

The equation that is used to compute net time is

$$\text{net time} = \text{elapsed time} - \text{nested time} - \text{wait time}$$

Elapsed time is the elapsed clock time this is required for processing an item. Nested time is the total of the elapsed times that are spent within the same thread to process other profiled items, such as substatements or declaration values. Wait time represents the time that a single thread is allocated but idle because it is waiting for other threads to perform the required processing. The total of net times can exceed the elapsed wall clock time when multiple threads are used.

PUT Statement

PUT [*put-items*] [@ | @@] **;**

The PUT statement writes text data to the current output file. The syntax of the PUT statement in PROC OPTMODEL is similar to the syntax of the PROC IML and DATA step PUT statements. The PUT statement contains a list of items that specify data for output and provide instructions for formatting the data.

The current output file is initially the SAS log. This can be overridden with the FILE statement. An output file can be closed with the CLOSEFILE statement.

Normally the PUT statement outputs the current line after processing all items. Final @ or @@ operators suppress this automatic line output and cause the current column position to be retained for use in the next PUT statement.

put-item can take any of the following forms.

identifier-expression [**=**] [*format*]
: outputs the value of the parameter or variable that is specified by the *identifier-expression*. The equal sign (=) causes a name for the location to be printed before each location value.

  Normally each item value is printed in a default format. Any leading and trailing blanks in the formatted value are removed, and the value is followed by a blank space. When an explicit format is specified, the value is printed within the width determined by the format.

*name***[*]** [**.***suffix*] [**=**] [*format*]
: outputs each defined location value for an array parameter. The array name is specified as in the *identifier-expression* form except that the index list is replaced by an asterisk (*). The equal sign (=) causes a name for the location to be printed before each location value along with the actual index values to be substituted for the asterisk.

  Each item value normally prints in a default format. Any leading and trailing blanks in the formatted value are removed, and the value is followed by a blank space. When an explicit format is specified, the value is printed within the width determined by the format.

(*expression* **)** [**=**] [*format*]
: outputs the value of the expression enclosed in parentheses. This produces similar results to the *identifier-expression* form except that the equal sign (=) uses the expression to form the name.

'*quoted-string*'
: copies the string to the output file.

@integer | *identifier-expression* | **(** *expression* **)** sets the absolute column position within the current line. The literal or expression value determines the new column position.

+*integer* |*identifier-expression*|**(** *expression* **)** sets the relative column position within the current line. The literal or expression value determines the amount to update the column position.

/

outputs the current line and moves to the first column of the next line.

PAGE

outputs any pending line data and moves to the top of the next page.

QUIT Statement

QUIT ;

The QUIT statement terminates the OPTMODEL execution. The statement is executed immediately, so it cannot be a nested statement. A QUIT statement is implied when a DATA or PROC statement is read.

READ DATA Statement

READ DATA *SAS-data-set* [**NOMISS**] **INTO** [[*set-name* **=**] **[** *read-key-columns* **]**] [*read-columns*] **;**

The READ DATA statement reads data from a SAS data set into PROC OPTMODEL parameter and variable locations. The arguments to the READ DATA statement are as follows:

SAS-data-set

specifies the input data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses.

set-name

specifies a set parameter in which to save the set of observation key values read from the input data set.

read-key-columns

provide the index values for array destinations.

read-columns

specify the data values to read and the destination locations.

The following example uses the READ DATA statement to copy data set variables j and k from the SAS data set indata into parameters of the same name. The READ= data set option specifies a password.

```
proc optmodel;
   number j, k;
   read data indata(read=secret) into j k;
```

Key Columns

If any *read-key-columns* are specified, then the READ DATA statement reads all observations from the input data set. If no *read-key-columns* are specified, then only the first observation of the data set is read. The data set is closed after reading the requested information.

Each *read-key-column* declares a local dummy parameter and specifies a data set variable that supplies the column value. The values of the specified data set variables from each observation are combined into a key

tuple. This combination is known as the *observation key*. The observation key is used to index array locations specified by the *read-columns* items. The observation key is expected to be unique for each observation read from the data set.

The syntax for a *read-key-column* is as follows:

name [**=** *source-name*] [/ *trim-option*]

A *read-key-column* creates a local dummy parameter named *name* that holds an element of the observation key tuple. The dummy parameter can be used in subsequent *read-columns* items to reference the element value. If a *source-name* is given, then it specifies the data set variable that supplies the value. Otherwise the source data set variable has the same name as the dummy parameter, *name*. Use the special data set variable name _N_ to refer to the number identification of the observations.

You can specify a *set-name* to save the set of observation keys into a set parameter. If the observation key consists of a single scalar value, then the set member type must match the scalar type. Otherwise the set member type must be a tuple with element types that match the corresponding observation key element types.

The READ DATA statement initially assigns an empty set to the target *set-name* parameter. As observations are read, a tuple for each observation key is added to the set. A set used to index an array destination in the *read-columns* can be read at the same time as the array values. Consider a data set, invdata, created by the following statements:

```
data invdata;
   input item $ invcount;
   datalines;
table 100
sofa  250
chair 80
;
```

The following statements read the data set invdata, which has two variables, item and invcount. The READ DATA statement constructs a set of inventory items, Items. At the same time, the parameter location invcount[item] is assigned the value of the data set variable invcount in the corresponding observation.

```
proc optmodel;
   set<string> Items;
   number invcount{Items};
   read data invdata into Items=[item] invcount;
   print invcount;
```

The output of these statements is shown in Figure 5.24.

Figure 5.24 READ DATA Statement: Key Column

[1]	invcount
chair	80
sofa	250
table	100

When observations are read, the values of data set variables are copied to parameter locations. Numeric values are copied unchanged. For character values, *trim-option* controls how leading and trailing blanks are processed. *trim-option* is ignored when the value type is numeric. Specify any of the following keywords for *trim-option*:

TRIM | **TR**
: removes leading and trailing blanks from the data set value. This is the default behavior.

LTRIM | **LT**
: removes only leading blanks from the data set value.

RTRIM | **RT**
: removes only trailing blanks from the data set value.

NOTRIM | **NT**
: copies the data set value with no changes.

Columns

read-columns specify data set variables to read and PROC OPTMODEL parameter locations to which to assign the values. The types of the input data set variables must match the types of the parameters. Array parameters can be implicitly or explicitly indexed by the observation key values.

Normally, missing values from the data set are assigned to the parameters that are specified in the *read-columns*. The NOMISS keyword suppresses the assignment of missing values, leaving the corresponding parameter locations unchanged. Note that the parameter location does not need to have a valid index in this case. This permits a single statement to read data into multiple arrays that have different index sets.

read-columns have the following forms:

identifier-expression [**=** *name* | **COL(** *name-expression* **)**] [/ *trim-option*]
: transfers an input data set variable to a target parameter or variable. *identifier-expression* specifies the target. If the *identifier-expression* specifies an array without an explicit index, then the observation key provides an implicit index. The name of the input data set variable can be specified with a *name* or a COL expression. Otherwise the data set variable name is given by the *name* part of the *identifier-expression*. For COL expressions, the string-valued *name-expression* is evaluated to determine the data set variable name. *trim-option* controls removal of leading and trailing blanks in the incoming data. For example, the following statements read the data set variables column1 and column2 from the data set exdata into the PROC OPTMODEL parameters p and q, respectively. The observation numbers in exdata are read into the set indx, which indexes p and q.

```
data exdata;
   input column1 column2;
   datalines;
1 2
3 4
;


proc optmodel;
   number n init 2;
   set<num> indx;
   number p{indx}, q{indx};
   read data exdata into
      indx=[_N_] p=column1 q=col("column"||n);
   print p q;
```

The output is shown in Figure 5.25.

Figure 5.25 READ DATA Statement: Identifier Expressions

[1]	p	q
1	1	2
2	3	4

{ *index-set* **} <** *read-columns* **>**

performs the transfers by iterating each column specified by <*read-columns*> for each member of the *index-set*. If there are n columns and m index set members, then $n \times m$ columns are generated. The dummy parameters from the index set can be used in the columns to generate distinct input data set variable names in the iterated columns, using COL expressions. The columns are expanded when the READ DATA statement is executed, before any observations are read. This form of *read-columns* cannot be nested. In other words, the following form of *read-columns* is NOT allowed:

{ *index-set* **} <** *{ index-set } < read-columns > >*

An example that demonstrates the use of the iterated column *read-option* follows.

You can use an iterated column *read-option* to read multiple data set variables into the same array. For example, a data set might store an entire row of array data in a group of data set variables. The following statements demonstrate how to read a data set that contains demand data divided by day:

```
data dmnd;
   input loc $ day1 day2 day3 day4 day5;
   datalines;
East 1.1 2.3 1.3 3.6 4.7
West 7.0 2.1 6.1 5.8 3.2
;

proc optmodel;
   set DOW = 1..5;  /* days of week, 1=Monday, 5=Friday */
   set<string> LOCS; /* locations */
   number demand{LOCS, DOW};
   read data dmnd
        into LOCS=[loc]
        {d in DOW} < demand[loc, d]=col("day"||d) >;
   print demand;
```

These statements read a set of demand variables named DAY1–DAY5 from each observation, filling in the two-dimensional array demand. The output is shown in Figure 5.26.

Figure 5.26 Demand Data

	demand				
	1	**2**	**3**	**4**	**5**
East	1.1	2.3	1.3	3.6	4.7
West	7.0	2.1	6.1	5.8	3.2

RESET OPTIONS Statement

RESET OPTIONS *options* **;**

RESET OPTION *options* **;**

The RESET OPTIONS statement sets PROC OPTMODEL option values or restores them to their defaults. Options can be specified by using the same syntax as in the PROC OPTMODEL statement. The RESET OPTIONS statement provides two extensions to the option syntax. If an option normally requires a value (specified with an equal sign (=) operator), then specifying the option name alone resets it to its default value. You can also specify an expression enclosed in parentheses in place of a literal value. See the section "OPTMODEL Options" on page 153 for an example.

The RESET OPTIONS statement can be placed inside loops or conditional statements. The statement is applied each time it is executed.

RESTORE Statement

RESTORE *identifier-list* **;**

The RESTORE statement adds a list of constraints, constraint arrays, or constraint array locations that were dropped by the DROP statement back into the solver model, or includes constraints in a problem where they were not previously present. The space-delimited *identifier-list* specifies the names of the constraints. Each constraint, constraint array, or constraint array location is named by an *identifier-expression*. An entire constraint array is restored if an *identifier-expression* omits the index from an array name. For example, the following statements declare a constraint array and then drop it:

```
con c{i in 1..4}: x[i] + y[i] <=1;
drop c;
```

The following statement restores the first constraint:

```
restore c[1];
```

The following statement restores the second and third constraints:

```
restore c[2] c[3];
```

If you want to restore all of the constraints, you can submit the following statement:

```
restore c;
```

SAVE MPS Statement

SAVE MPS *SAS-data-set* [**(OBJECTIVE** | **OBJ)** *name*] [**(NOOBJECTIVE** | **NOOBJ)**] **;**

The SAVE MPS statement saves the structure and coefficients for a linear programming model into a SAS data set. This data set can be used as input data for the OPTLP or OPTMILP procedure.

NOTE: The OPTMODEL presolver (see the section "Presolver" on page 145) is automatically bypassed so that the statement saves the original model *without* eliminating fixed variables, tightening bounds, and so on.

The *SAS-data-set* argument specifies the output data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses. The output data set uses the MPS format described in Chapter 17, "The MPS-Format SAS Data Set." The generated data set contains observations that define different parts of the linear program.

Variables, constraints, and objectives are referenced in the data set by using label text from the corresponding .label suffix value. The default text is based on the name in the model. See the section "Suffixes" on page 134 for more details. Labels are limited by default to 32 characters and are abbreviated to fit. You can change the maximum length for labels by using the MAXLABLEN= option. When needed, a programmatically generated number is added to labels to avoid duplication.

If the OBJECTIVE keyword is used, the objective *name* becomes the current problem objective. If the NOOBJECTIVE keyword is used or the current problem does not have an objective, then the data set includes a default constant zero objective. Otherwise, the current problem objective is included in the data set.

When an integer variable has been assigned a nondefault branching priority or direction, the MPS data set includes a BRANCH section. See Chapter 17, "The MPS-Format SAS Data Set," for more details.

The following statements show an example of the SAVE MPS statement. The model is specified using the OPTMODEL procedure. Then it is saved as the MPS data set MPSData, as shown in Figure 5.27. Next, PROC OPTLP is used to solve the resulting linear program.

```
proc optmodel;
   var x >= 0, y >= 0;
   con c: x >= y;
   con bx: x <= 2;
   con by: y <= 1;
   min obj=0.5*x-y;
   save mps MPSData;
quit;

proc optlp data=MPSData pout=PrimalOut dout=DualOut;
run;
```

Figure 5.27 The MPS Data Set Generated by SAVE MPS Statement

Obs	FIELD1	FIELD2	FIELD3	FIELD4	FIELD5	FIELD6
1	NAME		MPSData	.		.
2	ROWS			.		.
3	N	obj		.		.
4	G	c		.		.
5	L	bx		.		.
6	L	by		.		.
7	COLUMNS			.		.
8		x	obj	0.5	c	1
9		x	bx	1.0		.
10		y	obj	-1.0	c	-1
11		y	by	1.0		.
12	RHS			.		.
13		.RHS.	bx	2.0		.
14		.RHS.	by	1.0		.
15	ENDATA			.		.

SAVE QPS Statement

SAVE QPS *SAS-data-set* [**(OBJECTIVE** | **OBJ)** *name*] [**(NOOBJECTIVE** | **NOOBJ)]** **;**

The SAVE QPS statement saves the structure and coefficients for a quadratic programming model into a SAS data set. This data set can be used as input data for the OPTQP procedure.

NOTE: The OPTMODEL presolver (see the section "Presolver" on page 145) is automatically bypassed so that the statement saves the original model *without* eliminating fixed variables, tightening bounds, and so on.

The *SAS-data-set* argument specifies the output data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses. The output data set uses the QPS format described in Chapter 17. The generated data set contains observations that define different parts of the quadratic program.

Variables, constraints, and objectives are referenced in the data set by using label text from the corresponding .label suffix value. The default text is based on the name in the model. See the section "Suffixes" on page 134 for more details. Labels are limited by default to 32 characters and are abbreviated to fit. You can change the maximum length for labels by using the MAXLABLEN= option. When needed, a programmatically generated number is added to labels to avoid duplication.

If the OBJECTIVE keyword is used, the objective *name* becomes the current problem objective. If the NOOBJECTIVE keyword is used or the current problem does not have an objective, then the data set includes a default constant zero objective. Otherwise, the current problem objective is included in the data set. The quadratic coefficients of the objective function appear in the QSECTION section of the output data set.

The following statements show an example of the SAVE QPS statement. The model is specified using the OPTMODEL procedure. Then it is saved as the QPS data set QPSData, as shown in Figure 5.28. Next, PROC OPTQP is used to solve the resulting quadratic program.

```
proc optmodel;
   var x{1..2} >= 0;
      min z = 2*x[1] + 3 * x[2] + x[1]**2 + 10*x[2]**2
                  + 2.5*x[1]*x[2];
      con c1: x[1] - x[2] <= 1;
      con c2: x[1] + 2*x[2] >= 100;
      save qps QPSData;
quit;

proc optqp data=QPSData pout=PrimalOut dout=DualOut;
run;
```

Figure 5.28 QPS Data Set Generated by the SAVE QPS Statement

Obs	FIELD1	FIELD2	FIELD3	FIELD4	FIELD5	FIELD6
1	NAME		QPSData	.		.
2	ROWS			.		.
3	N	z		.		.
4	L	c1		.		.
5	G	c2		.		.
6	COLUMNS			.		.
7		x[1]	z	2.0	c1	1
8		x[1]	c2	1.0		.
9		x[2]	z	3.0	c1	-1
10		x[2]	c2	2.0		.
11	RHS			.		.
12		.RHS.	c1	1.0		.
13		.RHS.	c2	100.0		.
14	QSECTION			.		.
15		x[1]	x[1]	2.0		.
16		x[1]	x[2]	2.5		.
17		x[2]	x[2]	20.0		.
18	ENDATA			.		.

SOLVE Statement

SOLVE [**WITH** *solver*] [**(OBJECTIVE** | **OBJ)** *name*] [**(NOOBJECTIVE** | **NOOBJ)**] [**RELAXINT**] [/ *options*] **;**

The SOLVE statement invokes a PROC OPTMODEL solver. The current model is first resolved to the numeric form that is required by the solver. The resolved model and possibly the current values of any optimization variables are passed to the solver. After the solver finishes executing, the SOLVE statement prints a short table that shows a summary of results from the solver (see the section "ODS Table and Variable Names" on page 124) and updates the _OROPTMODEL_ macro variable.

Here are the arguments to the SOLVE statement:

solver

selects the named solver: CLP, LP, MILP, NETWORK, NLP, or QP (see corresponding chapters in this book for details). If you do not specify a WITH clause, PROC OPTMODEL chooses a solver that depends on the problem type. Table 5.8 lists the default solver for each problem type.[1]

[1] The OPTMODEL procedure never uses the network solver as a default. If the QP solver detects nonconvexity (nonconcavity) for a minimization (maximization) problem, then PROC OPTMODEL calls the NLP solver instead.

Table 5.8 Default Solvers and Algorithms in PROC OPTMODEL

Problem	Solver	Algorithm
Constraint programming	**CLP**	Constraint propagation and backtracking search
Linear programming	**LP**	Dual simplex
Mixed integer linear programming	**MILP**	Branch-and-cut
General nonlinear programming	**NLP**	Interior point NLP
Quadratic programming	**QP**	Interior point QP

name

specifies the objective to use. This sets the current objective for the problem. You can abbreviate the OBJECTIVE keyword as OBJ. If this argument is not specified, then the problem objective is unchanged.

NOOBJECTIVE requests that the solver ignore the current objective for the problem and use a constant zero objective instead. This keyword enables the solver to process the current model as a feasibility problem. You can abbreviate the NOOBJECTIVE keyword as NOOBJ.

RELAXINT requests that any integral variables be relaxed to be continuous. RELAXINT can be used with linear and nonlinear problems in addition to any solver.

options specifies solver options. You can specify solver options directly only when you use the WITH clause. A list of the options available with the solver is provided in the individual chapters that describe each solver. Table 5.9 lists the available option types. You can use an expression in parentheses in place of a literal option value for numeric and keyword options. A string expression is matched to a keyword. OPTMODEL parameters that are changed by the solver must be specified by a parameter or array option.

Table 5.9 Solver Option Types

Type	Syntax	Example
Boolean	*option* | NO*option*	**solve with nlp / NOMULTISTART;**
Keyword	*option=name*	**solve with lp / ALGORITHM=PS;**
Numeric	*option=number*	**solve with nlp / OPTTOL=1E–4;**
Parameter	*option=identifier-expression*	**solve with network / links=(INCLUDE=LINKS) concomp;**
Array	*option=array-name* [*.suffix*]	**solve with network / links=(WEIGHT=WEIGHT) tsp;**

The SOLVE statement uses the value of the predeclared _SOLVER_OPTIONS_ and _*solver*_OPTIONS_ string parameters to provide default solver options. Any options that are specified by these parameters are added before options that are specified in the SOLVE statement, with options from _SOLVER_OPTIONS_ appearing first. These options are included even when the SOLVE statement does not contain a WITH clause to specify a solver; in this case, *solver* is the name of the default solver as shown in Table 5.8.

Initially the predeclared string parameters _SOLVER_OPTIONS_ and _*solver*_OPTIONS_ (for each solver) are empty strings, but you can assign them. You must use keywords or literal values to specify option values in these strings. Redundant white space is allowed. For example, the following statements set up some simple defaults:

```
_SOLVER_OPTIONS_ = "MAXTIME = 600"; /* options for all solvers */
_LP_OPTIONS_ = "PRESOLVER=AGGRESSIVE"; /* options for LP solver */
```

Optimization techniques that use initial values obtain them from the current values of the optimization variables unless the NOINITVAR option is specified. When the solver finishes executing, the current value of each optimization variable is replaced by the optimal value found by the solver. These values can then be used as the initial values for subsequent solver invocations.

NOTE: If a solver fails, any currently pending statement is stopped and processing continues with the next complete statement read from the input. For example, if a SOLVE statement that is enclosed in a DO group (see the section "DO Statement" on page 66) fails, then the subsequent statements in the group are not executed and processing resumes at the point immediately following the DO group. Neither an infeasible result, an unbounded result, nor reaching an iteration limit is considered to be a solver failure.

NOTE: The information that appears in the macro variable _OROPTMODEL_ (see the section "Macro Variable _OROPTMODEL_" on page 161) varies by solver.

NOTE: The RELAXINT keyword is applied immediately before the problem is passed to the solver, after any processing by the PROC OPTMODEL presolver. So the problem presented to the solver might not be equivalent to the one produced by setting the .RELAX suffix of all variables to a nonzero value. In particular, the bounds of integer variables are still adjusted to be integral, and PROC OPTMODEL's presolver might use integrality to tighten bounds further.

STOP Statement

STOP ;

The STOP statement halts the execution of all statements that contain it, including DO statements and other control or looping statements. Execution continues with the next top-level source statement. The following statements demonstrate a simple use of the STOP statement:

```
proc optmodel;
   number i, j;
   do i = 1..5;
      do j = 1..4;
         if i = 3 and j = 2 then stop;
      end;
   end;
   print i j;
```

The output is shown in Figure 5.29.

Figure 5.29 STOP Statement: Output

i	j
3	2

When the counters i and j reach 3 and 2, respectively, the STOP statement terminates both loops. Execution continues with the PRINT statement.

SUBMIT Statement

SUBMIT *arguments* [/ *options*] **;**

SAS statements **;**

ENDSUBMIT ;

The SUBMIT statement allows SAS code to be executed before PROC OPTMODEL processing continues. For example, you can use the SUBMIT statement to invoke other SAS procedures to perform analysis or to display results. The following statements use PROC SORT to order a list of nodes by decreasing priority; the nodes can be used for further processing:

```
proc optmodel;
   set<str> NODES;
   num priority{NODES};

   /* set up priority data... */

   /* sort nodes by descending priority */
   create data temppri from [id] priority;
   submit;
      proc sort;
         by descending priority;
      run;
   endsubmit;

   /* load nodes by priority */
   str nodesByPri{i in 1..card(NODES)};
   read data temppri into [_n_] nodesByPri=id;

   /* use the sorted list... */
```

The SUBMIT statement must appear as the last or only statement on a line. It is followed by lines of SAS statements, terminated by the ENDSUBMIT statement on a line of its own. The SAS statements between the SUBMIT and ENDSUBMIT statements are referred to as a *SUBMIT block*. The SUBMIT block is sent to the SAS language processor each time the SUBMIT statement is executed.

The SUBMIT block can include SAS global statements and procedure and invocations. Macros are not expanded until the SUBMIT block is executed. So you can change macro variables to modify the behavior of the SUBMIT block each time it is processed.

The *arguments* list specifies macro variables to initialize in the SUBMIT block environment before the SUBMIT block is executed. List items are separated by spaces. Each of the *arguments* takes one of the following forms:

name
: copies the value of the PROC OPTMODEL parameter *name* to the macro variable that has the same name.

name **=** *identifier-expression*
: copies the value of the PROC OPTMODEL parameter specified by *identifier-expression* to the macro variable *name*.

name **=** *number* | “*string*” | ‘*string*’
: copies the value of the specified *number* or *string* constant to the macro variable *name*.

name **=** *(expression)*
: copies the result of evaluating *expression* to the macro variable *name*.

The following statements use a SUBMIT argument to modify the output each time the SUBMIT block is invoked:

```
for {i in 1..5}
   submit a=i;
   %put Value of a is &a..;
   endsubmit;
```

The *options* in the SUBMIT statement are used to retrieve status information after a SUBMIT block is executed. Each item in the space-delimited *options* list has one of the following forms:

OK = *identifier-expression*
: specifies a PROC OPTMODEL numeric parameter location, *identifier-expression*, that is updated to indicate the success of the SUBMIT block execution. The location is set to 1 if execution is successful or 0 if errors are detected. PROC OPTMODEL continues execution when the SUBMIT block encounters errors only if the OK= option is specified.

OUT [**=**] *output-argument*
: specifies a single *output-argument* for retrieving macro variable values from the SUBMIT block environment after each execution of the block.

OUT [**=**] **(** *output-argument* **)**
: specifies a list of space-delimited *output-arguments* for retrieving macro variable values from the SUBMIT block environment after the block is executed.

Each *output-argument* item specifies a macro variable to copy out of the SUBMIT block environment after the block is executed. Each item takes one of the following two forms:

identifier-expression
: copies the macro variable specified by the *name* portion of the *identifier-expression* into the PROC OPTMODEL parameter location specified by *identifier-expression*.

identifier-expression **=** *name*
: copies the macro variable specified by *name* into the PROC OPTMODEL parameter location specified by *identifier-expression*.

The following statements show how to use the *options* in the SUBMIT statement to retrieve the result of a SUBMIT block execution:

```
proc optmodel;
   num success, syscc;
   submit / OK = success out syscc;
      data example;
         set notfound;
             j = i*i;
             run;
   endsubmit;
   print success syscc;
```

The DATA step fails, so the success parameter is set to 0 and syscc is set to the error code in the &SYSCC macro variable. The output is shown in Figure 5.30.

Figure 5.30 SUBMIT Statement Error Handling

success	syscc
0	1012

NOTE: The SUBMIT block runs in an environment that is nested in the environment that the OPTMODEL procedure is running in. Resources from the PROC OPTMODEL environment are initially visible in the nested environment. However, the nested environment can have its own local values for options, LIBNAME librefs, FILENAME filerefs, titles, footnotes, and macros. For example, the nested environment has its own global macro scope, which can hide macros visible in the outer environment. The *output-arguments* of the SUBMIT statement *options* can retrieve the values of macros defined in this scope.

NOTE: A SUBMIT block can reset the ODS environment of the OPTMODEL procedure. For example, the ODS SELECT and EXCLUDE lists could be cleared after the SUBMIT block executes.

NOTE: A SUBMIT statement can appear only in open code. An error message is displayed if the SUBMIT statement is read from a macro. You can avoid this limitation by placing the SUBMIT statement, SUBMIT block, and ENDSUBMIT in a separate file and by using the %INCLUDE statement to include the file in the macro.

UNFIX Statement

UNFIX *identifier-list* [**= (** *expression* **)**] **;**

The UNFIX statement reverses the effect of FIX statements. The solver can vary the specified variables, variable arrays, or variable array locations specified by *identifier-list*. The *identifier-list* consists of one or more variable names separated by spaces.

Each variable name in the *identifier-list* is an *identifier expression* (see the section "Identifier Expressions" on page 102). The UNFIX statement affects an entire variable array if the identifier expression omits the index from an array name. The *expression* specifies a new initial value that is stored in each element of the *identifier-list*.

The following example demonstrates the UNFIX command:

```
proc optmodel;
   var x{1..3};
   fix x;       /* fixes entire array to 0 */
   unfix x[1];  /* x[1] can now be varied again */
   unfix x[2] = 2;  /* x[2] is given an initial value 2 */
                 /* and can be varied now */
   unfix x;  /* all x indices can now be varied */
```

After the following statements are executed, the variables x[1] and x[2] are not fixed. They each hold the value 4. The variable x[3] is fixed at a value of 2.

```
proc optmodel;
   var x{1..3} init 2;
   num a = 1;
   fix x;
   unfix x[1] x[2]=(a+3);
```

USE PROBLEM Statement

USE PROBLEM *identifier-expression* **;**

The USE PROBLEM programming statement makes the problem specified by the *identifier-expression* be the current problem. If the problem has not been previously used, the problem is created using the PROBLEM declaration corresponding to the name. The problem must have been previously declared.

Details: OPTMODEL Procedure

Named Parameters

In the example described in the section "An Unconstrained Optimization Example" on page 30, all the numeric constants that describe the behavior of the objective function were specified directly in the objective expression. This is a valid way to formulate the objective expression. However, in many cases it is inconvenient to specify the numeric constants directly. Direct specification of numeric constants can also hide the structure of the problem that is being solved. The objective expression text would need to be modified when the numeric values in the problem change. This can be very inconvenient with large models.

In PROC OPTMODEL, you can create named numeric values that behave as constants in expressions. These named values are called *parameters*. You can write an expression by using mnemonic parameter names in place of numeric literals. This produces a clearer formulation of the optimization problem. You can easily modify the values of parameters, define them in terms of other parameters, or read them from a SAS data set.

The model from this same example can be reformulated in a more general polynomial form, as follows:

```
data coeff;
   input c_xx c_x c_y c_xy c_yy;
   datalines;
   1 -1  -2 -1 1
   ;
proc optmodel;
   var x, y;
   number c_xx, c_x, c_y, c_xy, c_yy;
   read data coeff into c_xx c_x c_y c_xy c_yy;
   min z=c_xx*x**2 + c_x*x + c_y*y + c_xy*x*y + c_yy*y**2;
   solve;
```

These statements read the coefficients from a data set, COEFF. The NUMBER statement declares the parameters. The READ DATA statement reads the parameters from the data set. You can apply this model easily to coefficients that you have generated by various means.

Indexing

Many models have large numbers of variables or parameters that can be categorized into families of similar purpose or behavior. Such families of items can be compactly represented in PROC OPTMODEL by using indexing. You can use indexing to assign each item in such families to a separate value location.

PROC OPTMODEL indexing is similar to array indexing in the DATA step, but it is more flexible. Index values can be numbers or strings, and are not required to fit into some rigid sequence. PROC OPTMODEL indexing is based on index sets, described further in the section "Index Sets" on page 103. For example, the following statement declares an indexed parameter:

```
number p{1..3};
```

The construct that follows the parameter name p, "{1..3}," is a simple index set that uses a range expression (see "Range Expression" on page 109). The index set contains the numeric members 1, 2, and 3. The parameter has distinct value locations for each of the index set members. The first such location is referenced as p[1], the second as p[2], and the third as p[3].

The following statements show an example of indexing:

```
proc optmodel;
   number p{1..3};
   p[1]=5;
   p[2]=7;
   p[3]=9;
   put p[*]=;
```

The preceding statements produce a line such as the one shown in Figure 5.31 in the log.

Figure 5.31 Indexed Parameter Output

```
p[1]=5 p[2]=7 p[3]=9
```

Index sets can also specify local dummy parameters. A dummy parameter can be used as an operand in the expressions that are controlled by the index set. For example, the assignment statements in the preceding statements could be replaced by an initialization in the parameter declaration, as follows:

```
number p{i in 1..3} init 3 + 2*i;
```

The initialization value of the parameter location p[1] is evaluated with the value of the local dummy parameter i equal to 1. So the initialization expression 3 + 2*i evaluates to 5. Similarly for location p[2], the value of i is 2 and the initialization expression evaluates to 7.

The OPTMODEL modeling language supports aggregation operators that combine values of an expression where a local dummy parameter (or parameters) ranges over the members of a set. For example, the SUM aggregation operator combines expression values by adding them together. The following statements output 21, since p[1] + p[2] + p[3] = 5 + 7 + 9 = 21:

```
proc optmodel;
   number p{i in 1..3} init 3 + 2*i;
   put (sum{i in 1..3} p[i]);
```

Aggregation operators like SUM are especially useful in objective expressions because they can combine a large number of similar expressions into a compact representation. As an example, the following statements define a trivial least squares problem:

```
proc optmodel;
   number n init 100000;
   var x{1..n};
   min z = sum{i in 1..n}(x[i] - log(i))**2;
   solve;
```

The objective function in this case is

$$z = \sum_{i=1}^{n} (x_i - \log i)^2$$

Effectively, the objective expression expands to the following large expression:

```
min z = (x[1] - log(1))**2
      + (x[2] - log(2))**2
      . . .
      + (x[99999] - log(99999))**2
      + (x[100000] - log(100000))**2;
```

Even though the problem has 100,000 variables, the aggregation operator SUM enables a compact objective expression.

NOTE: PROC OPTMODEL classifies as mathematically impure any function that returns a different value each time it is called. The RAND function, for example, falls into this category. PROC OPTMODEL disallows impure functions inside array index sets, objectives, and constraint expressions. The values of expressions that are specified in the declaration of a parameter are resolved in a nondeterministic order during threaded problem generation. Therefore, the values are also nondeterministic when these expressions use impure functions.

Types

In PROC OPTMODEL, parameters and expressions can have numeric or character values. These correspond to the elementary types named NUMBER and STRING, respectively. The NUMBER type is the same as the SAS data set numeric type. The NUMBER type includes support for missing values. The STRING type corresponds to the SAS character type, except that strings can have lengths up to a maximum of 65,534 characters (versus 32,767 for SAS character-type variables). The length for a STRING can change as needed. The NUMBER and STRING types together are called the *scalar types*. You can abbreviate the type names as NUM and STR, respectively.

PROC OPTMODEL also supports set types for parameters and expressions. Sets represent collections of values of a member type, which can be a NUMBER, a STRING, or a vector of scalars (the latter is called a *tuple* and described in the following paragraphs). Members of a set all have the same member type. Members that have the same value are stored only once. For example, PROC OPTMODEL stores the set 2, 2, 2 as the set 2.

Specify a set of numbers with SET<NUMBER>. Similarly, specify a set of strings as SET<STRING>.

A set can also contain a collection of tuples, all of the same fixed length. A *tuple* is an ordered collection that contains a fixed number of elements. Each element in a tuple contains a scalar value. In PROC OPTMODEL, tuples of length 1 are equivalent to scalars. Two tuples have equal values if the elements at corresponding positions in each tuple have the same value. Within a set of tuples, the element type at a particular position in each tuple is the same for all set members. The element types are part of the set type. For example, the following statement declares parts as a set of tuples that have a string in the first element position and a number in the second element position and then initializes its elements to be <R 1>, <R 2>, <C 1>, and <C 2>.

```
set<string,number> parts = /<R 1> <R 2> <C 1> <C 2>/;
```

To create a compact model, use sets to take advantage of the structure of the problem being modeled. For example, a model might contain various values that specify attributes for each member of a group of suppliers. You could create a set that contains members that represent each supplier. You can then model the attribute values by using arrays that are indexed by members of the set.

The section "Parameters" on page 98 has more details and examples.

Names

Names are used in the OPTMODEL modeling language to refer to various entities such as parameters or variables. Names must follow the usual rules for SAS names. Names can be up to 32 characters long and are not case sensitive. They must be declared before they are used.

Avoid declarations with names that begin with an underscore (_). These names can have special uses in PROC OPTMODEL.

Parameters

In the OPTMODEL modeling language, parameters are named locations that hold constant values. Parameter declarations specify the parameter type followed by a list of parameter names to declare. For example, the following statement declares numeric parameters named a and b:

```
number a, b;
```

Similarly, the following statements declare a set s of strings, a set n of numbers, and a set sn of tuples:

```
set<string> s;
set<number> n;
set<string, number> sn;
```

You can assign values to parameters in various ways. A parameter can be assigned a value with an assignment statement. For example, the following statements assign values to the parameter s, n, and sn in the preceding declaration:

```
s = {'a', 'b', 'c'};
n = {1, 2, 3};
sn = {<'a',1>, <'b',2>, <'c',3>};
```

Parameter values can also be assigned using a READ DATA statement (see the section "READ DATA Statement" on page 82).

A parameter declaration can provide an explicit value. To specify the value, follow the parameter name with an equal sign (=) and an expression. The value expression can be written in terms of other parameters. The declared parameter takes on a new value each time a parameter that is used in the expression changes. This automatic value update is shown in the following example:

```
proc optmodel;
   number pi=4*atan(1);
   number r;
   number circum=2*pi*r;
   r=1;
   put circum;       /* prints 6.2831853072 */
   r=2;
   put circum;       /* prints 12.566370614 */
```

The automatic update of parameter values makes it easy to perform "what if" analysis since, after the solver finds a solution, you can change parameters and reinvoke the solver. You can easily examine the effects of the changes on the optimal values.

If you declare a set parameter that has only the SET type specifier, then the element type is determined from the initialization expression. If the initialization expression is omitted or if the expression is an empty set, then the set type defaults to SET<NUMBER>. For example, the following statement implicitly declares `s1` as a set of numbers:

```
set s1;
```

The following statement declares `s2` as a set of strings:

```
set s2 = {'A'};
```

You can declare an array parameter by following the parameter name with an index set specification (see the section "Index Sets" on page 103). For example, declare an array of 10 numbers as follows:

```
number c{1..10};
```

Individual locations of a parameter array can be referred to with an indexing expression. For example, you can refer to the third location of parameter `c` as `c[3]`. Array index sets *cannot* be specified using a function such as RAND that returns a different value each time it is called.

Parameter names must be declared before they are used. Nonarray names become available at the end of the parameter declaration item. Array names become available after the index set specification. The latter case permits some forms of recursion in the optional initialization expression that can be supplied for a parameter.

You do not need to assign values to parameters before they are referenced. Most information in PROC OPTMODEL is stored symbolically and resolved when necessary. Values are resolved in certain statements. For example, PROC OPTMODEL resolves a parameter used in the objective during the execution of a SOLVE statement. If no value is available during resolution, then an error is diagnosed.

Expressions

Expressions are grouped into three categories based on the types of values they can produce: logical, set, and scalar (that is, numeric or character).

Logical expressions test for a Boolean (true or false) condition. As in the DATA step, logical operators produce a value equal to either 0 or 1. A value of 0 represents a false condition, while a value of 1 represents a true condition.

Logical expression operators are not allowed in certain contexts due to syntactic considerations. For example, in the VAR statement a logical operator might indicate the start of an option. Enclose a logical expression

in parentheses to use it in such contexts. The difference is illustrated by the output (Figure 5.32) of the following statements, where two variables, x and y, are declared with initial values. The PRINT statement and the EXPAND statement are used to check the initial values and the variable bounds, respectively.

```
proc optmodel;
   var x init 0.5 >= 0 <= 1;
   var y init (0.5 >= 0) <= 1;
   print x y;
   expand;
```

Figure 5.32 Logical Expression in the VAR Statement

x	y
0.5	1

```
Var x >= 0 <= 1
Var y <= 1
```

Contexts that expect a logical expression also accept numeric expressions. In such cases zero or missing values are interpreted as false, and all nonzero nonmissing numeric values are interpreted as true.

Set expressions return a set value. PROC OPTMODEL supports a number of operators that create and manipulate sets. See the section "OPTMODEL Expression Extensions" on page 104 for a description of the various set expressions. Index-set syntax is described in the section "Index Sets" on page 103.

Scalar expressions are similar to the expressions in the DATA step except for PROC OPTMODEL extensions. PROC OPTMODEL provides an IF expression (described in the section "IF-THEN/ELSE Expression" on page 106). String lengths are assigned dynamically, so there is generally no padding or truncation of string values.

Table 5.10 shows the expression operators from lower to higher precedence (a higher precedence is given a larger number). Operators that have higher precedence are applied in compound expressions before operators that have lower precedence. The table also gives the order of evaluation that is applied when multiple operators of the same precedence are used together. Operators available in both PROC OPTMODEL and the DATA step have compatible precedences, except that in PROC OPTMODEL the NOT operator has a lower precedence than the relational operators. This means that, for example, `NOT 1 < 2` is equal to `NOT (1 < 2)` (which is 0), rather than `(NOT 1) < 2` (which is 1).

Table 5.10 Expression Operator Table

Precedence	Associativity	Operator	Alternates
Logic Expression Operators			
1	Left to right	`OR`	`| !`
2	Unary	`OR{`*index-set*`}`	
		`AND{`*index-set*`}`	
3	Left to right	`AND`	`&`
4	Unary	`NOT`	`~ ^ ¬`
5	Left to right	`<`	`LT`
		`>`	`GT`
		`<=`	`LE`
		`>=`	`GE`
		`=`	`EQ`
		`~=`	`NE ^= ¬=`

Table 5.10 (continued)

Precedence	Associativity	Operator	Alternates
6	Left to right	`IN`	
		`NOT IN`	
7	Left to right	`WITHIN`	
		`NOT WITHIN`	
Set Expression Operators			
11		`IF l THEN s1 ELSE s2`	
12	Left to right	`UNION`	
		`DIFF`	
		`SYMDIFF`	
13	Unary	`UNION{`*index-set*`}`	
14	Left to right	`INTER`	
15	Unary	`INTER{`*index-set*`}`	
16	Left to right	`CROSS`	
17	Unary	`SETOF{`*index-set*`}`	
	Right to left	`..`	`TO`
		`.. e BY`	`TO e BY`
Scalar Expression Operators			
21		`IF l THEN e`	
		`IF l THEN e1 ELSE e2`	
22	Left to right	`||`	`!!`
23	Left to right	`+ -`	
24	Unary	`SUM{`*index-set*`}`	
		`PROD{`*index-set*`}`	
		`MIN{`*index-set*`}`	
		`MAX{`*index-set*`}`	
25	Left to right	`* /`	
26	Unary	`+ -`	
	Right to left	`><`	
		`<>`	
		`**`	`^`

Primary expressions are the individual operands that are combined using the expression operators. Simple primary expressions can represent constants or named parameter and variable values. More complex primary expressions can be used to call functions or construct sets.

Table 5.11 Primary Expression Table

Expression	Description
identifier-expression	Parameter/variable reference; see the section "Identifier Expressions" on page 102
name (*arg-list*)	Function call; *arg-list* is 0 or more expressions separated by commas
n	Numeric constant
. or **.***c*	Missing value constant
"*string*" or '*string*'	String constant

Table 5.11 (continued)

Expression	Description
{ *member-list* **}**	Set constructor; *member-list* is 0 or more scalar expressions or tuple expressions separated by commas
{ *index-set* **}**	Index set expression; returns the set of all index set members
/ *members* /	Set literal expression; compactly specifies a simple set value
(*expression* **)**	Expression enclosed in parentheses
< *expr-list* **>**	Tuple expression; used with set operations; contains one or more scalar expressions separated by commas

Identifier Expressions

Use an *identifier-expression* to refer to a variable, objective, constraint, parameter or problem location in expressions or initializations. This is the syntax for *identifier-expressions*:

name [**[** *expression-1* [**,** *. . . expression-n*] **]**] [**.** *suffix* [**[** *expression* **]**]]

To refer to a location in an array, follow the array *name* with a list of scalar expressions in square brackets ([ ]). The expression values are compared to the index set that was used to declare *name*. If there is more than one expression, then the values are formed into a tuple. The expression values for a valid array location must match a member of the array's index set. For example, the following statements define a parameter array A that has two valid indices that match the tuples <1,2> and <3,4>:

```
proc optmodel;
   set<number, number> ISET = {<1,2>, <3,4>};
   number A{ISET};
   a[1,2] = 0;  /* OK */
   a[3,2] = 0;  /* invalid index */
```

The first assignment is valid with this definition of the index set, but the second fails because <3,2> is not a member of the set parameter ISET.

Specify a *suffix* to refer to auxiliary locations for variables or objectives. For more information, see the section "Suffixes" on page 134. Certain suffixes can be followed by a numeric index *expression* that selects a particular solution saved by the SOLVE statement. For more information about solution indices, see the section "Multiple Solutions" on page 151.

Function Expressions

Most functions that can be invoked from the DATA step or the %SYSFUNC macro can be used in PROC OPTMODEL expressions. Certain functions are specific to the DATA step and cannot be used in PROC OPTMODEL. Functions specific to the DATA step include these:

- functions in the LAG, DIF, and DIM families

- functions that access the DATA step program data vector
- functions that access symbol attributes

The CALL statement can invoke SAS library subroutines. These subroutines can read and update the values of the parameters and variables that are used as arguments. See the section "CALL Statement" on page 54 for an example.

OPTMODEL arrays can be passed to SAS library functions and subroutines using the argument syntax:

OF *array-name***[*]** [**.** *suffix*]

The *array-name* is the name of an array symbol. The optional *suffix* allows auxiliary values to be referenced, as described in section "Suffixes" on page 134.

The OF argument form is resolved into a sequence of arguments, one for each index in the array. The array elements appear in order of the array's index set. The OF array form is a compact alternative to listing the array elements explicitly.

As an example, the following statements use the CALL SORTN function to sort the elements of a numeric array:

```
proc optmodel;
   number original{i in 1..8} = sin(i);
   number sorted{i in 1..8} init original[i];
   call sortn(of sorted[*]);
   print original sorted;
```

The output is shown in Figure 5.33. Eight arguments are passed to the SORTN routine. The original column shows the original order, and the sorted column has the sorted order.

Figure 5.33 Sorting Using an OF Array Argument

[1]	original	sorted
1	0.84147	-0.95892
2	0.90930	-0.75680
3	0.14112	-0.27942
4	-0.75680	0.14112
5	-0.95892	0.65699
6	-0.27942	0.84147
7	0.65699	0.90930
8	0.98936	0.98936

Index Sets

An index set represents a set of combinations of members from the component set expressions. The index set notation is used in PROC OPTMODEL to describe collections of valid array indices and to specify sets of values with which to perform an operation. Index sets can declare local dummy parameters and can further restrict the set of combinations by a selection expression.

In an index-set specification, the index set consists of one or more *index-set-items* that are separated by commas. Each *index-set-item* can include local dummy parameter declarations. An optional selection expression follows the list of *index-set-items*. The following syntax, which describes an index set, usually appears in braces ({}):

index-set-item [**,** *. . . index-set-item*] [**:** *logic-expression*]

index-set-item has these forms:

set-expression
name **IN** *set-expression*
< *name-1* [**,** *. . . name-n*] **>** **IN** *set-expression*

Names that precede the IN keyword in *index-set-items* declare local dummy parameter names. Dummy parameters correspond to the dummy index variables in mathematical expressions. For example, the following statements output the number 385:

```
proc optmodel;
   put (sum{i in 1..10} i**2);
```

The preceding statements evaluate this summation:

$$\sum_{i=1}^{10} i^2 = 385$$

In both the statements and the summation, the index name is *i*.

The last form of *index-set-item* in the list can be modified to use the SLICE expression implicitly. See the section "More on Index Sets" on page 159 for details.

Array index sets cannot be defined using functions that return different values each time the functions are called. See the section "Indexing" on page 95 for details.

OPTMODEL Expression Extensions

PROC OPTMODEL defines several new types of expressions for the manipulation of sets. Aggregation operators combine values of an expression that is evaluated over the members of an index set. Other operators create new sets by combining existing sets, or they test relationships between sets. PROC OPTMODEL also supports an IF expression operator that can conditionally evaluate expressions. These and other such expressions are described in this section.

AND Aggregation Expression

AND { *index-set* **}** *logic-expression*

The AND aggregation operator evaluates the logical expression *logic-expression* jointly for each member of the index set *index-set*. The index set enumeration finishes early if the *logic-expression* evaluation produces a false value (zero or missing). The expression returns 0 if a false value is found or returns 1 otherwise. The following statements demonstrate both a true and a false result:

```
proc optmodel;
   put (and{i in 1..5} i < 10); /* returns 1 */
   put (and{i in 1..5} i NE 3); /* returns 0 */
```

CARD Function

CARD (*set-expression* **)**

The CARD function returns the number of members of its set operand. For example, the following statements produce the output 3 since the set has 3 members:

```
proc optmodel;
   put (card(1..3));
```

CROSS Expression

set-expression-1 **CROSS** *set-expression-2*

The CROSS expression returns the crossproduct of its set operands. The result is the set of tuples formed by concatenating the tuple value of each member of the left operand with the tuple value of each member of the right operand. Scalar set members are treated as tuples of length 1. The following statements demonstrate the CROSS operator:

```
proc optmodel;
   set s1 = 1..2;
   set<string> s2 = {'a', 'b'};
   set<number, string> s3=s1 cross s2;
   put 's3 is ' s3;
   set<number, string, number> s4 = s3 cross 4..5;
   put 's4 is ' s4;
```

This code produces the output in Figure 5.34.

Figure 5.34 CROSS Expression Output

```
s3 is {<1,'a'>,<1,'b'>,<2,'a'>,<2,'b'>}
s4 is {<1,'a',4>,<1,'a',5>,<1,'b',4>,<1,'b',5>,<2,'a',4>,<2,'a',5>,<2,'b',4>,<2,
'b',5>}
```

DIFF Expression

set-expression-1 **DIFF** *set-expression-2*

The DIFF operator returns a set that contains the set difference of the left and right operands. The result set contains values that are members of the left operand but not members of the right operand. The operands must have compatible set types. The following statements evaluate and print a set difference:

```
proc optmodel;
   put ({1,3} diff {2,3}); /* outputs {1} */
```

IF-THEN/ELSE Expression

IF *logic-expression* **THEN** *expression-2* [**ELSE** *expression-3*]

The IF-THEN/ELSE expression evaluates the logical expression *logic-expression* and returns the result of evaluating the second or third operand expression according to the logical test result. If the *logic-expression* is true (nonzero and nonmissing), then the result of evaluating *expression-2* is returned. If the *logic-expression* is false (zero or missing), then the result of evaluating *expression-3* is returned. The other subexpression that is not selected is not evaluated.

An ELSE clause is matched during parsing with the nearest IF-THEN clause that does not have a matching ELSE. The ELSE clause can be omitted for numeric expressions; the resulting IF-THEN is handled as if a default ELSE 0 clause were supplied.

Use the IF-THEN/ELSE expression to handle special cases in models. For example, an inventory model based on discrete time periods might require special handling for the first or last period. In the following example the initial inventory for the first period is assumed to be fixed:

```
proc optmodel;
   number T;
   var inv{1..T}, order{1..T};
   number sell{1..T};
   number inv0;
   . . .
   /* balance inventory flow */
   con iflow{i in 1..T}:
       inv[i] = order[i] - sell[i] +
       if i=1 then inv0 else inv[i-1];
   . . .
```

The IF-THEN/ELSE expression in the example models the initial inventory for a time period *i*. Usually the inventory value is the inventory at the end of the previous period, but for the first time period the inventory value is given by the inv0 parameter. The iflow constraints are linear because the IF-THEN/ELSE test subexpression does not depend on variables and the other subexpressions are linear.

IF-THEN/ELSE can be used as either a set expression or a scalar expression. The type of expression depends on the subexpression between the THEN and ELSE keywords. The type used affects the parsing of the subexpression that follows the ELSE keyword because the set form has a lower operator precedence. For example, the following two expressions are equivalent because the numeric IF-THEN/ELSE has a higher precedence than the range operator (..):

```
IF logic THEN 1 ELSE 2 .. 3

(IF logic THEN 1 ELSE 2) .. 3
```

But the set form of IF-THEN/ELSE has lower precedence than the range expression operator. So the following two expressions are equivalent:

```
IF logic THEN 1 .. 2 ELSE 3 .. 4

IF logic THEN (1 .. 2) ELSE (3 .. 4)
```

The IF-THEN and IF-THEN/ELSE operators always have higher precedence than the logic operators. So, for example, the following two expressions are equivalent:

```
IF logic THEN numeric1 < numeric2

(IF logic THEN numeric1) < numeric2
```

It is best to use parentheses when in doubt about precedence.

IN Expression

expression **IN** *set-expression*

expression **NOT IN** *set-expression*

The IN expression returns 1 if the value of the left operand is a member of the right operand set. Otherwise, the IN expression returns 0. The NOT IN operator logically negates the returned value. Unlike the DATA step, the right operand is an arbitrary set expression. The left operand can be a tuple expression. The following example demonstrates the IN and NOT IN operators:

```
proc optmodel;
   set s = 1..10;
   put (5 in s);          /* outputs 1 */
   put (-1 not in s);     /* outputs 1 */
   set<num, str> t = {<1,'a'>, <2,'b'>, <2,'c'>};
   put (<2, 'b'> in t);   /* outputs 1 */
   put (<1, 'b'> in t);  /* outputs 0 */
```

Index Set Expression

{ *index-set* **}**

The index set expression returns the set of members of an index set. This expression is distinguished from a set constructor (see the section "Set Constructor Expression" on page 110) because it contains a list of set expressions.

The following statements use an index set with a selection expression that excludes the value 3:

```
proc optmodel;
   put ({i in 1..5 : i NE 3}); /* outputs {1,2,4,5} */
```

INTER Expression

set-expression-1 **INTER** *set-expression-2*

The INTER operator returns a set that contains the intersection of the left and right operands. This is the set that contains values that are members of both operand sets. The operands must have compatible set types.

The following statements evaluate and print a set intersection:

```
proc optmodel;
   put ({1,3} inter {2,3}); /* outputs {3} */
```

INTER Aggregation Expression

INTER { *index-set* **}** *set-expression*

The INTER aggregation operator evaluates the *set-expression* for each member of the index set *index-set*. The result is the set that contains the intersection of the set of values that were returned by the *set-expression* for each member of the index set. An empty index set causes an expression evaluation error.

The following statements use the INTER aggregation operator to compute the value of {1,2,3,4} ∩ {2,3,4,5} ∩ {3,4,5,6}:

```
proc optmodel;
   put (inter{i in 1..3} i..i+3); /* outputs {3,4} */
```

MAX Aggregation Expression

MAX { *index-set* **}** *expression*

The MAX aggregation operator evaluates the numeric expression *expression* for each member of the index set *index-set*. The result is the maximum of the values that are returned by the *expression*. Missing values are handled with the SAS numeric sort order; a missing value is treated as smaller than any nonmissing value. If the index set is empty, then the result is the negative number that has the largest absolute value representable on the machine.

The following example produces the output `0.5`:

```
proc optmodel;
   put (max{i in 2..5} 1/i);
```

MIN Aggregation Expression

MIN { *index-set* **}** *expression*

The MIN aggregation operator evaluates the numeric expression *expression* for each member of the index set *index-set*. The result is the minimum of the values that are returned by the *expression*. Missing values are handled with the SAS numeric sort order; a missing value is treated as smaller than any nonmissing value. If the index set is empty, then the result is the largest positive number representable on the machine.

The following example produces the output `0.2`:

```
proc optmodel;
   put (min{i in 2..5} 1/i);
```

OR Aggregation Expression

OR { *index-set* **}** *logic-expression*

The OR aggregation operator evaluates the logical expression *logic-expression* for each member of the index set *index-set*. The index set enumeration finishes early if the *logic-expression* evaluation produces a true value (nonzero and nonmissing). The result is 1 if a true value is found, or 0 otherwise. The following statements demonstrate both a true and a false result:

```
proc optmodel;
   put (or{i in 1..5} i = 2); /* returns 1 */
   put (or{i in 1..5} i = 7); /* returns 0 */
```

PROD Aggregation Expression

PROD { *index-set* **}** *expression*

The PROD aggregation operator evaluates the numeric expression *expression* for each member of the index set *index-set*. The result is the product of the values that are returned by the *expression*. This operator is analogous to the $\prod$ operator used in mathematical notation. If the index set is empty, then the result is 1.

The following example uses the PROD operator to evaluate a factorial:

```
proc optmodel;
   number n = 5;
   put (prod{i in 1..n} i); /* outputs 120 */
```

Range Expression

expression-1 .. expression-n [**BY** *expression*]

The range expression returns the set of numbers from the specified arithmetic progression. The sequence proceeds from the left operand value up to the right operand limit. The increment between numbers is 1 unless a different value is specified with a BY clause. If the increment is negative, then the progression is from the left operand down to the right operand limit. The result can be an empty set.

For compatibility with the DATA step iterative DO loop construct, the keyword TO can substitute for the range (..) operator.

The limit value is not included in the resulting set unless it belongs in the arithmetic progression. For example, the following range expression does not include 30:

```
proc optmodel;
   put (10..30 by 7); /* outputs {10,17,24} */
```

The actual numbers that the range expression "f..l by i" produces are in the arithmetic sequence

$$f, f+i, f+2i, \ldots, f+ni$$

where

$$n = \left\lfloor \frac{l-f}{i} + \sqrt{\epsilon} \right\rfloor$$

and ϵ represents the relative machine precision. The limit is adjusted to avoid arithmetic roundoff errors.

PROC OPTMODEL represents the set specified by a range expression compactly when the value is stored in a parameter location, used as a set operand of an IN or NOT IN expression, used by an iterative DO loop, or used in an index set. For example, the following expression is evaluated efficiently:

```
999998.5 IN 1..1000000000
```

Set Constructor Expression

{ [*expression-1* [**,** . . . *expression-n*]] **}**

The set constructor expression returns the set of the expressions in the member list. Duplicated values are added to the set only once. A warning message is produced when duplicates are detected. The constructor expression consists of zero or more subexpressions of the same scalar type or of tuple expressions that match in length and in element types.

The following statements output a three-member set and warn about the duplicated value 2:

```
proc optmodel;
   put ({1,2,3,2}); /* outputs {1,2,3} */
```

The following example produces a three-member set of tuples, using PROC OPTMODEL parameters and variables. The output is displayed in Figure 5.35.

```
proc optmodel;
   number m = 3, n = 4;
   var x{1..4} init 1;
   string y = 'c';
   put ({<'a', x[3]>, <'b', m>, <y, m/n>});
```

Figure 5.35 Set Constructor Expression Output

```
{<'a',1>,<'b',3>,<'c',0.75>}
```

Set Literal Expression

/ *members* **/**

The set literal expression provides compact specification of simple set values. It is equivalent in function to the set constructor expression but minimizes typing for sets that contain numeric and string constant values. The set members are specified by *members*, which are literal values. As with the set constructor expression, each member must have the same type.

The following statement specifies a simple numeric set:

```
/1 2.5 4/
```

The set contains the members `1`, `2.5`, and `4`. A string set could be specified as follows:

```
/Miami 'San Francisco' Seattle 'Washington, D.C.'/
```

This set contains the strings `'Miami'`, `'San Francisco'`, `'Seattle'`, and `'Washington, D.C.'`. You can specify string values in set literals without quotation marks when the text follows the rules for a SAS name. Strings that begin with a digit or contain blanks or other special characters must be specified with quotation marks.

Specify tuple members of a set by enclosing the tuple elements within angle brackets (<*elements*>). The tuple elements can be specified with numeric and string literals. The following example includes the tuple elements `<'New York', 4.5>` and `<'Chicago', -5.7>`:

```
/<'New York' 4.5> <Chicago -5.7>/
```

SETOF Aggregation Expression

SETOF { *index-set* **}** *expression*

The SETOF aggregation operator evaluates the expression *expression* for each member of the index set *index-set*. The result is the set that is formed by collecting the values returned by the operand expression. The operand can be a tuple expression. For example, the following statements produce a set of tuples of numbers with their squared and cubed values:

```
proc optmodel;
   put (setof{i in 1..3}<i, i*i, i**3>);
```

Figure 5.36 shows the displayed output.

Figure 5.36 SETOF Aggregation Expression Output

```
{<1,1,1>,<2,4,8>,<3,9,27>}
```

SLICE Expression

SLICE (< *element-1*, . . . *element-n* **>** , *set-expression* **)**

The SLICE expression produces a new set by selecting members in the operand set that match a pattern tuple. The pattern tuple is specified by the element list in angle brackets. Each *element* in the pattern tuple must specify a numeric or string expression. The expressions are used to match the values of the corresponding elements in the operand set member tuples. You can also specify an *element* by using an asterisk (*). The sequence of element values that correspond to asterisk positions in each matching tuple is combined into a tuple of the result set. At least one asterisk *element* must be specified.

The following statements demonstrate the SLICE expression:

```
proc optmodel;
   put (slice(<1,*>, {<1,3>, <1,0>, <3,1>}));
   put (slice(<*,2,*>, {<1,2,3>, <2,4,3>, <2,2,5>}));
```

These statements produce the output in Figure 5.37.

Figure 5.37 SLICE Expression Output

```
{3,0}
{<1,3>,<2,5>}
```

For the first PUT statement, <1,*> matches set members <1,3> and <1,0> but not <3,1>. The second element of each matching set tuple, corresponding to the asterisk element, becomes the value of the resulting set member. In the second PUT statement, the values of the first and third elements of the operand set member tuple are combined into a two-position tuple in the result set.

The following statements use the SLICE expression to help compute the transitive closure of a set of tuples representing a relation by using Warshall's algorithm. In these statements the set parameter dep represents a direct dependency relation.

```
proc optmodel;
   set<str,str> dep = {<'B','A'>, <'C','B'>, <'D','C'>};
   set<str,str> cl;
   set<str> cn;
   cl = dep;
   cn =  (setof{<i,j> in dep} i) inter (setof{<i,j> in dep} j);
   for {node in cn}
       cl = cl union (slice(<*,node>,cl) cross slice(<node,*>,cl));
   put cl;
```

The local dummy parameter node in the FOR statement iterates over the set cn of possible intermediate nodes that can connect relations transitively. At the end of each FOR iteration, the set parameter cl contains all tuples from the original set in addition to all transitive tuples found in the current or previous iterations.

The output in Figure 5.38 includes the indirect and direct transitive dependencies from the set dep.

Figure 5.38 Warshall's Algorithm Output

```
{<'B','A'>,<'C','B'>,<'D','C'>,<'C','A'>,<'D','B'>,<'D','A'>}
```

A special form of *index-set-item* uses the SLICE expression implicitly. See the section "More on Index Sets" on page 159 for details.

SUM Aggregation Expression

SUM { *index-set* **}** *expression*

The SUM aggregation operator evaluates the numeric expression *expression* for each member in the index set *index-set*. The result is the sum of the values that are returned by the *expression*. If the index set is empty, then the result is 0. This operator is analogous to the $\sum$ operator that is used in mathematical notation. The following statements demonstrate the use of the SUM aggregation operator:

```
proc optmodel;
   put (sum {i in 1..10} i);  /* outputs 55 */
```

SYMDIFF Expression

set-expression-1 **SYMDIFF** *set-expression-2*

The SYMDIFF expression returns the symmetric set difference of the left and right operands. The result set contains values that are members of either the left or right operand but are not members of both operands. The operands must have compatible set types.

The following example demonstrates a symmetric difference:

```
proc optmodel;
   put ({1,3} symdiff {2,3}); /* outputs {1,2} */
```

Tuple Expression

< *expression-1*, . . . *expression-n* **>**

A tuple expression represents the value of a member in a set of tuples. Each scalar subexpression inside the angle brackets represents the value of a tuple element. This form is used only with IN, SETOF, and set constructor expressions.

The following statements demonstrate the tuple expression:

```
proc optmodel;
   put (<1,2,3> in setof{i in 1..2}<i,i+1,i+2>);
   put ({<1,'a'>, <2,'b'>} cross {<3,'c'>, <4,'d'>});
```

The first PUT statement checks whether the tuple <1, 2, 3> is a member of a set of tuples. The second PUT statement outputs the crossproduct of two sets of tuples that are constructed by the set constructor.

These statements produce the output in Figure 5.39.

Figure 5.39 Tuple Expression Output

```
1
{<1,'a',3,'c'>,<1,'a',4,'d'>,<2,'b',3,'c'>,<2,'b',4,'d'>}
```

UNION Expression

set-expression-1 **UNION** *set-expression-2*

The UNION expression returns the set union of the left and right operands. The result set contains values that are members of either the left or right operand. The operands must have compatible set types. The following example performs a set union:

```
proc optmodel;
   put ({1,3} union {2,3}); /* outputs {1,3,2} */
```

UNION Aggregation Expression

UNION { *index-set* **}** *set-expression*

The UNION aggregation expression evaluates the *set-expression* for each member of the index set *index-set*. The result is the set union of the values that are returned by the *set-expression*. If the index set is empty, then the result is an empty set.

The following statements demonstrate a UNION aggregation. The output is the value of {1,2,3,4} ∪ {2,3,4,5} ∪ {3,4,5,6}.

```
proc optmodel;
   put (union{i in 1..3} i..i+3); /* outputs {1,2,3,4,5,6} */
```

WITHIN Expression

set-expression-1 **WITHIN** *set-expression-2*

set-expression **NOT WITHIN** *set-expression*

The WITHIN expression returns 1 if the left operand set is a subset of the right operand set and returns 0 otherwise. (That is, the operator returns true if every member of the left operand set is a member of the right operand set.) The NOT WITHIN form logically negates the result value. The following statements demonstrate the WITHIN and NOT WITHIN operators:

```
proc optmodel;
   put ({1,3} within {2,3});     /* outputs 0 */
   put ({1,3} not within {2,3}); /* outputs 1 */
   put ({1,3} within {1,2,3});   /* outputs 1 */
```

Conditions of Optimality

Linear Programming

A standard linear program has the following formulation:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{array}$$

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of decision variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the matrix of constraints
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)

This formulation is called the primal problem. The corresponding dual problem (see the section "Dual Values" on page 138) is

$$\begin{array}{ll} \text{maximize} & \mathbf{b}^T\mathbf{y} \\ \text{subject to} & \mathbf{A}^T\mathbf{y} \leq \mathbf{c} \\ & \mathbf{y} \geq 0 \end{array}$$

where $\mathbf{y} \in \mathbb{R}^m$ is the vector of dual variables.

The vectors $\mathbf{x}$ and $\mathbf{y}$ are optimal to the primal and dual problems, respectively, only if there exist primal slack variables $\mathbf{s} = \mathbf{A}\mathbf{x} - \mathbf{b}$ and dual slack variables $\mathbf{w} = \mathbf{A}^T\mathbf{y} - \mathbf{c}$ such that the following *Karush-Kuhn-Tucker (KKT) conditions* are satisfied:

$$\begin{array}{rcl} \mathbf{A}\mathbf{x} + \mathbf{s} & = & \mathbf{b}, \quad \mathbf{x} \geq 0, \quad \mathbf{s} \geq 0 \\ \mathbf{A}^T\mathbf{y} + \mathbf{w} & = & \mathbf{c}, \quad \mathbf{y} \geq 0, \quad \mathbf{w} \geq 0 \\ \mathbf{s}^T\mathbf{y} & = & 0 \\ \mathbf{w}^T\mathbf{x} & = & 0 \end{array}$$

The first line of equations defines primal feasibility, the second line of equations defines dual feasibility, and the last two equations are called the complementary slackness conditions.

Nonlinear Programming

To facilitate discussion of optimality conditions in nonlinear programming, you write the general form of nonlinear optimization problems by grouping the equality constraints and inequality constraints. You also write all the general nonlinear inequality constraints and bound constraints in one form as "$\geq$" inequality constraints. Thus, you have the following formulation:

$$\begin{array}{ll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & f(x) \\ \text{subject to} & c_i(x) = 0, \quad i \in \mathcal{E} \\ & c_i(x) \geq 0, \quad i \in \mathcal{I} \end{array}$$

where $\mathcal{E}$ is the set of indices of the equality constraints, $\mathcal{I}$ is the set of indices of the inequality constraints, and $m = |\mathcal{E}| + |\mathcal{I}|$.

A point x is *feasible* if it satisfies all the constraints $c_i(x) = 0, i \in \mathcal{E}$ and $c_i(x) \geq 0, i \in \mathcal{I}$. The feasible region $\mathcal{F}$ consists of all the feasible points. In unconstrained cases, the feasible region $\mathcal{F}$ is the entire $\mathbb{R}^n$ space.

A feasible point x^* is a *local solution* of the problem if there exists a neighborhood $\mathcal{N}$ of x^* such that

$$f(x) \geq f(x^*) \quad for\ all\ x \in \mathcal{N} \cap \mathcal{F}$$

Further, a feasible point x^* is a *strict local solution* if strict inequality holds in the preceding case; that is,

$$f(x) > f(x^*) \quad for\ all\ x \in \mathcal{N} \cap \mathcal{F}$$

A feasible point x^* is a *global solution* of the problem if no point in $\mathcal{F}$ has a smaller function value than $f(x^*)$; that is,

$$f(x) \geq f(x^*) \quad for\ all\ x \in \mathcal{F}$$

Unconstrained Optimization

The following conditions hold true for unconstrained optimization problems:

- **First-order necessary conditions**: If x^* is a local solution and $f(x)$ is continuously differentiable in some neighborhood of x^*, then

  $$\nabla f(x^*) = 0$$

- **Second-order necessary conditions**: If x^* is a local solution and $f(x)$ is twice continuously differentiable in some neighborhood of x^*, then $\nabla^2 f(x^*)$ is positive semidefinite.

- **Second-order sufficient conditions**: If $f(x)$ is twice continuously differentiable in some neighborhood of x^*, $\nabla f(x^*) = 0$, and $\nabla^2 f(x^*)$ is positive definite, then x^* is a strict local solution.

Constrained Optimization

For constrained optimization problems, the *Lagrangian function* is defined as follows:

$$L(x, \lambda) = f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i c_i(x)$$

where $\lambda_i, i \in \mathcal{E} \cup \mathcal{I}$, are called *Lagrange multipliers*. $\nabla_x L(x, \lambda)$ is used to denote the gradient of the Lagrangian function with respect to x, and $\nabla_x^2 L(x, \lambda)$ is used to denote the Hessian of the Lagrangian function with respect to x. The active set at a feasible point x is defined as

$$\mathcal{A}(x) = \mathcal{E} \cup \{i \in \mathcal{I} : c_i(x) = 0\}$$

You also need the following definition before you can state the first-order and second-order necessary conditions:

- **Linear independence constraint qualification and regular point**: A point x is said to satisfy the *linear independence constraint qualification* if the gradients of active constraints

  $$\nabla c_i(x), \quad i \in \mathcal{A}(x)$$

  are linearly independent. Such a point x is called a *regular point*.

You now state the theorems that are essential in the analysis and design of algorithms for constrained optimization:

- **First-order necessary conditions**: Suppose that x^* is a local minimum and also a regular point. If $f(x)$ and $c_i(x), i \in \mathcal{E} \cup \mathcal{I}$, are continuously differentiable, there exist Lagrange multipliers $\lambda^* \in \mathbb{R}^m$ such that the following conditions hold:

  $$\begin{aligned} \nabla_x L(x^*, \lambda^*) = \nabla f(x^*) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i^* \nabla c_i(x^*) &= 0 \\ c_i(x^*) &= 0, \quad i \in \mathcal{E} \\ c_i(x^*) &\geq 0, \quad i \in \mathcal{I} \\ \lambda_i^* &\geq 0, \quad i \in \mathcal{I} \\ \lambda_i^* c_i(x^*) &= 0, \quad i \in \mathcal{I} \end{aligned}$$

  The preceding conditions are often known as the *Karush-Kuhn-Tucker conditions*, or *KKT conditions* for short.

- **Second-order necessary conditions**: Suppose that x^* is a local minimum and also a regular point. Let λ^* be the Lagrange multipliers that satisfy the KKT conditions. If $f(x)$ and $c_i(x), i \in \mathcal{E} \cup \mathcal{I}$, are twice continuously differentiable, the following conditions hold:

  $$z^T \nabla_x^2 L(x^*, \lambda^*) z \geq 0$$

  for all $z \in \mathbb{R}^n$ that satisfy

  $$\nabla c_i(x^*)^T z = 0, \quad i \in \mathcal{A}(x^*)$$

- **Second-order sufficient conditions**: Suppose there exist a point x^* and some Lagrange multipliers λ^* such that the KKT conditions are satisfied. If

  $$z^T \nabla_x^2 L(x^*, \lambda^*) z > 0$$

  for all $z \in \mathbb{R}^n$ that satisfy

  $$\nabla c_i(x^*)^T z = 0, \quad i \in \mathcal{A}(x^*)$$

  then x^* is a strict local solution.

  Note that the set of all such z's forms the null space of the matrix $\left[\nabla c_i(x^*)^T\right]_{i \in \mathcal{A}(x^*)}$. Thus, you can search for strict local solutions by numerically checking the Hessian of the Lagrangian function projected onto the null space. For a rigorous treatment of the optimality conditions, see Fletcher (1987) and Nocedal and Wright (1999).

Data Set Input/Output

You can use the CREATE DATA and READ DATA statements to exchange PROC OPTMODEL data with SAS data sets. The statements can move data into and out of PROC OPTMODEL parameters and variables. For example, the following statements use a CREATE DATA statement to save the results from an optimization into a data set:

```
proc optmodel;
   var x;
   min z = (x-5)**2;
   solve;
   create data optdata from xopt=x z;
```

These statements write a single observation into the data set OPTDATA. The data set contains two variables, xopt and z, and the values contain the optimized values of the PROC OPTMODEL variable x and objective z, respectively. The statement "xopt=x" renames the variable x to xopt.

The group of values held by a data set variable in different observations of a data set is referred to as a *column*. The READ DATA and CREATE DATA statements specify a set of columns for a data set and define how data are to be transferred between the columns and PROC OPTMODEL parameters.

Columns in square brackets ([ ]) are handled specially. Such columns are called *key columns*. Key columns specify element values that provide an implicit index for subsequent array columns. The following example uses key columns with the CREATE DATA statement to write out variable values from an array:

```
proc optmodel;
   set LOCS = {'New York', 'Washington', 'Boston'};  /* locations */
   set DOW = 1..7;  /* day of week */
   var s{LOCS, DOW} init 1;
   create data soldata from [location day_of_week]={LOCS, DOW} sale=s;
```

In this case the optimization variable s is initialized to a value of 1 and is indexed by values from the set parameters LOCS and DOW. The output data set contains an observation for each combination of values in these sets. The output data set contains three variables, location, day_of_week, and sale. The data set variables location and day_of_week save the index element values for the optimization variable s that is written in each observation. The data set created is shown in Figure 5.40.

Figure 5.40 Data Sets Created

Data Set: SOLDATA

Obs	location	day_of_week	sale
1	New York	1	1
2	New York	2	1
3	New York	3	1
4	New York	4	1
5	New York	5	1
6	New York	6	1
7	New York	7	1
8	Washington	1	1
9	Washington	2	1
10	Washington	3	1
11	Washington	4	1
12	Washington	5	1
13	Washington	6	1
14	Washington	7	1
15	Boston	1	1
16	Boston	2	1
17	Boston	3	1
18	Boston	4	1
19	Boston	5	1
20	Boston	6	1
21	Boston	7	1

Note that the key columns in the preceding example do not name existing PROC OPTMODEL variables. They create new local dummy parameters, location and day_of_week, in the same manner as dummy parameters in index sets. These local parameters can be used in subsequent columns. For example, the following statements demonstrate how to use a key column value in an expression for a later column value:

```
proc optmodel;
   create data tab
          from [i]=(1..10)
          Square=(i*i) Cube=(i*i*i);
```

These statements create a data set that has 10 observations that hold squares and cubes of the numbers from 1 to 10. The key column variable here is named i and is explicitly assigned the values from 1 to 10, while the data set variables Square and Cube hold the square and cube, respectively, of the corresponding value of i.

In the preceding example the key column values are simply the numbers from 1 to 10. The value is the same as the observation number, so the variable i is redundant. You can remove the data set variable for a key column via the DROP data set option, as follows:

```
proc optmodel;
   create data tab2 (drop=i)
          from [i] =(1..10)
          Square=(i*i) Cube=(i*i*i);
```

The local parameters declared by key columns receive their values in various ways. For a READ DATA statement, the key column values come from the data set variables for the column. In a CREATE DATA statement, the values can be defined explicitly, as shown in the previous example. Otherwise, the CREATE DATA statement generates a set of values that combines the index sets of array columns that need implicit indexing. The statements that produce the output in Figure 5.40 demonstrate implicit indexing.

Use a suffix ("Suffixes" on page 134) to read or write auxiliary values, such as variable bounds or constraint duals. For example, consider the following statements:

```
data pdat;
   input p $ maxprod cost;
   datalines;
ABQ    12  0.7
MIA     9  0.6
CHI    14  0.5
run;

proc optmodel;
   set<string> plants;
   var prod{plants} >= 0;
   number cost{plants};
   read data pdat into plants=[p] prod.ub=maxprod cost;
```

The statement "plants=[p]" in the READ DATA statement declares p as a key column and instructs PROC OPTMODEL to store the set of plant names from the data set variable p into the set parameter plants. The statement assigns the upper bound for the variable prod indexed by p to be the value of the data set variable maxprod. The cost parameter location indexed by p is also assigned to be the value of the data set variable cost.

The target variables prod and cost in the preceding example use implicit indexing. Indexing can also be performed explicitly. The following version of the READ DATA statement makes the indices explicit:

```
read data pdat into plants=[p] prod[p].ub=maxprod cost[p];
```

Explicit indexing is useful when array indices need to be transformed from the key column values in the data set. For example, the following statements reverse the order in which elements from the data set are stored in an array:

```
data abcd;
   input letter $ @@;
   datalines;
a b c d
;

proc optmodel;
   set<num> subscripts=1..4;
   string  letter{subscripts};
   read data abcd into [_N_] letter[5-_N_];
   print letter;
```

The output from this example appears in Figure 5.41.

Figure 5.41 READ DATA Statement: Explicit Indexing

[1]	letter
1	d
2	c
3	b
4	a

The following example demonstrates the use of explicit indexing to save sequential subsets of an array in individual data sets:

```
data revdata;
   input month rev @@;
   datalines;
1 200 2 345 3 362 4 958
5 659 6 804 7 487 8 146
9 683 10 732 11 652 12 469
;

proc optmodel;
   set m = 1..3;
   var revenue{1..12};
   read data revdata into [_N_] revenue=rev;
   create data qtr1 from [month]=m revenue[month];
   create data qtr2 from [month]=m revenue[month+3];
   create data qtr3 from [month]=m revenue[month+6];
   create data qtr4 from [month]=m revenue[month+9];
```

Each CREATE DATA statement generates a data set that represents one quarter of the year. Each data set contains the variables month and revenue. The data set qtr2 is shown in Figure 5.42.

Figure 5.42 CREATE DATA Statement: Explicit Indexing

Obs	month	revenue
1	1	958
2	2	659
3	3	804

Data set names and options that are used in PROC OPTMODEL statements can also be generated dynamically, by using an expression in parentheses to specify the name and options. The expression must evaluate to a string value with the following form:

[*libref* **.**] *member* [**(** *options* **)**]

Dynamic data set names are useful for processing data sets inside looping statements such as a FOR statement, particularly when the loop contains a SOLVE statement. You can replace the multiple CREATE DATA statements in the previous example with a single statement in a loop:

```
proc optmodel;
   set m = 1..3;
   var revenue{1..12};
   read data revdata into [_N_] revenue=rev;
   for {q in 1..4}
      create data ("qtr" || q)
      from [month]=m revenue[month+(q-1)*3];
```

Control Flow

Most of the control flow statements in PROC OPTMODEL are familiar to users of the DATA step or the IML procedure. PROC OPTMODEL supports the IF statement, DO blocks, the iterative DO statement, the DO WHILE statement, and the DO UNTIL statement. You can also use the CONTINUE, LEAVE, and STOP statements to modify control flow.

PROC OPTMODEL adds the FOR statement. This statement is similar in operation to an iterative DO loop. However, the iteration is performed over the members of an index set. This form is convenient for iteration over all the locations in an array, since the valid array indices are also defined using an index set. For example, the following statements initialize the array parameter A, indexed by i and j, to random values sampled from a normal distribution with mean 0 and variance 1:

```
proc optmodel;
   set R=1..10;
   set C=1..5;
   number A{R, C};
   for {i in R, j in C}
       A[i, j]=rannor(-1);
```

The FOR statement provides a convenient way to perform a statement such as the preceding assignment statement for each member of a set.

Formatted Output

PROC OPTMODEL provides two primary means of producing formatted output. The PUT statement provides output of data values with detailed format control. The PRINT statement handles arrays and produces formatted output in tabular form.

The PUT statement is similar in syntax to the PUT statement in the DATA step and in PROC IML. The PUT statement can output data to the SAS log, the SAS listing, or an external file. Arguments to the PUT statement specify the data to output and provide instructions for formatting. The PUT statement provides enough control to create reports within PROC OPTMODEL. However, typically the PUT statement is used to produce output for debugging or to quickly check data values.

The following example demonstrates some features of the PUT statement:

```
proc optmodel;
   number a=1.7, b=2.8;
   set s={a,b};
   put a b;         /* list output */
   put a= b=;       /* named output */
   put 'Value A: ' a 8.1 @30 'Value B: ' b 8.; /* formatted */
   string str='Ratio (A/B) is:';
   put str (a/b);  /* strings and expressions */
   put s=;          /* named set output */
```

These statements produce the output in Figure 5.43.

Figure 5.43 PUT Statement Output

```
1.7 2.8
a=1.7 b=2.8
Value A:      1.7           Value B:        3
Ratio (A/B) is: 0.6071428571
s={1.7,2.8}
```

The first PUT statement demonstrates list output. The numeric data values are output in a default format, BEST12., with leading and trailing blanks removed. A blank space is inserted after each data value is output. The second PUT statement uses the equal sign (=) to request that the variable name be output along with the regular list output.

The third PUT statement demonstrates formatted output. It uses the @ operator to position the output in a specific column. This style of output can be used in report generation. The format specification "8." causes the displayed value of parameter b to be rounded.

The fourth PUT statement shows the output of a string value, str. It also outputs the value of an expression enclosed in parentheses. The final PUT statement outputs a set along with its name.

The default destination for PUT statement output is the SAS log. The FILE and CLOSEFILE statements can be used to send output to the SAS listing or to an external data file. Multiple files can be open at the same time. The FILE statement selects the current destination for PUT statement output, and the CLOSEFILE statement closes the corresponding file. See the section "FILE Statement" on page 71 for more details.

The PRINT statement is designed to output numeric and string data in the form of tables. The PRINT statement handles the details of formatting automatically. However, the output format can be overridden by PROC OPTMODEL options and through Output Delivery System (ODS) facilities.

The PRINT statement can output array data in a table form that contains a row for each combination of array index values. This form uses columns to display the array index values for each row and uses other columns to display the value of each requested data item. The following statements demonstrate the table form:

```
proc optmodel;
   number square{i in 0..5} = i*i;
   number recip{i in 1..5} = 1/i;
   print square recip;
```

The PRINT statement produces the output in Figure 5.44.

Figure 5.44 PRINT Statement Output (List Form)

[1]	square	recip
0	0	
1	1	1.00000
2	4	0.50000
3	9	0.33333
4	16	0.25000
5	25	0.20000

The first table column, labeled "[1]," contains the index values for the parameters square and recip. The columns that are labeled "square" and "recip" contain the parameter values for each array index. For example, the last row corresponds to the index 5 and the value in the last column is 0.2, which is the value of recip[5].

Note that the first row of the table contains no value in the recip column. Parameter location recip[0] does not have a valid index, so no value is printed. The PRINT statement does not display variables that are undefined or have invalid indices. This permits arrays that have similar indexing to be printed together. The sets of defined indices in the arrays are combined to generate the set of indices shown in the table.

Also note that the PRINT statement has assigned formats and widths that differ between the square and recip columns. The PRINT statement assigns a default fixed-point format to produce the best overall output for each data column. The format that is selected depends on the PDIGITS= and PWIDTH= options.

The PDIGITS= and PWIDTH= options specify the desired significant digits and formatted width, respectively. If the range of magnitudes is large enough that no suitable format can be found, then the data item is displayed in scientific format. The table in the preceding example displays the last column with five decimal places in order to display the five significant digits that were requested by the default PDIGITS= value. The square column, on the other hand, does not need any decimal places.

The PRINT statement can also display two-dimensional arrays in matrix form. If the list following the PRINT statement contains only a single array that has two index elements, then the array is displayed in matrix form when it is sufficiently dense (otherwise the display is in table form). In this form the left-most column contains the values of the first index element. The remaining columns correspond to and are labeled by the values of the second index element. The following statements print an example of matrix form:

```
proc optmodel;
   set R=1..6;
   set C=1..4;
   number a{i in R, j in C} = 10*i+j;
   print a;
```

The PRINT statement produces the output in Figure 5.45.

Figure 5.45 PRINT Statement Output (Matrix Form)

	a			
	1	**2**	**3**	**4**
1	11	12	13	14
2	21	22	23	24
3	31	32	33	34
4	41	42	43	44
5	51	52	53	54
6	61	62	63	64

In the example the first index element ranges from 1 to 6 and corresponds to the table rows. The second index element ranges from 1 to 4 and corresponds to the table columns. Array values can be found based on the row and column values. For example, the value of parameter a[3,2] is 32. This location is found in the table in the row labeled "3" and the column labeled "2."

ODS Table and Variable Names

PROC OPTMODEL assigns a name to each table it creates. You can use these names to reference the table when you use the Output Delivery System (ODS) to select tables and create output data sets. The names of tables common to all solvers are listed in Table 5.12. Some solvers can generate additional tables; see the individual solver chapters for more information. For more information about ODS, see *SAS Output Delivery System: User's Guide*.

Table 5.12 ODS Tables Produced in PROC OPTMODEL

ODS Table Name	Description	Statement/Option
CoforPerfInfo	List of COFOR statement performance options and their values	COFOR
DerivMethods	List of derivatives used by the solver, including the method of computation	SOLVE
OptStatistics	Solver-dependent description of the resources required for solution, including function evaluations and solver time	SOLVE
PrintTable	Specified parameter or variable values	PRINT
ProblemSummary	Description of objective, variables, and constraints	SOLVE
ProfileInfo	Detailed timing of statements and declarations	PROFILE
SolutionSummary	Overview of solution, including solver-dependent solution quality values	SOLVE
SolverOptions	List of solver options and their values	SOLVE
PerformanceInfo	List of solver performance options and their values	SOLVE
Timing	Detailed solution timing	PERFORMANCE / DETAILS

To guarantee that ODS output data sets contain information from all executed statements, use the PERSIST= option in the ODS OUTPUT statement. For details, see *SAS Output Delivery System: User's Guide*. **NOTE:** The SUBMIT statement resets ODS SELECT and EXCLUDE lists.

Table 5.13 lists the variable names of the preceding tables used in the ODS template of the OPTMODEL procedure.

Table 5.13 Variable Names for the ODS Tables Produced in PROC OPTMODEL

ODS Table Name	Variables
CoforPerfInfo	Label1, cValue1, and nValue1
DerivMethods	Label1, cValue1, and nValue1
OptStatistics	Label1, cValue1, and nValue1
PrintTable (matrix form)	ROW, COL1 – COL*n*
PrintTable (table form)	COL1 – COL*n*, *identifier-expression(_suffix)*
ProblemSummary	Label1, cValue1, and nValue1
ProfileInfo	Item, Line, Column, Count, NetTime, WaitTime, and PctTime
SolutionSummary	Label1, cValue1, and nValue1
SolverOptions	Label1, cValue1, nValue1, cValue2, and nValue2
PerformanceInfo	Label1, cValue1, and nValue1
Timing	Label1, cValue1, nValue1, cValue2, and nValue2

The PRINT statement produces an ODS table named PrintTable. The variable names that are used depend on the display format used. See the section "Formatted Output" on page 121 for details about choosing the display format.

For the PRINT statement with table format, the columns that display array indices are named COL1–COL*n*, where *n* is the number of index elements. Columns that display values from identifier expressions are named using the expression's name and suffix. The identifier name becomes the output variable name if no suffix is used. Otherwise the variable name is formed by appending an underscore (_) and the suffix to the identifier name. Columns that display the value of expressions are named COL*n*, where *n* is the column number in the table.

For the PRINT statement with matrix format, the first column has the variable name ROW. The remaining columns are named COL1–COL*n*, where *n* is the number of distinct column indices. When an ODS table displays values from identifier expressions, a label is generated based on the expression's name and suffix, as described for column names in the case of table format.

The PRINTLEVEL= option controls the ODS tables produced by the SOLVE and COFOR statements. When PRINTLEVEL=0, the statements produce no ODS tables. When PRINTLEVEL=1, the SOLVE statement produces the ODS tables ProblemSummary, SolutionSummary, and PerformanceInfo. When PRINTLEVEL=2, the SOLVE statement produces the ODS tables ProblemSummary, SolverOptions, DerivMethods, SolutionSummary, OptStatistics, and PerformanceInfo. When PRINTLEVEL=1 or 2, the COFOR statement produces the ODS table CoforPerfInfo.

The PERFORMANCE statement controls additional ODS tables that can be produced by the SOLVE and COFOR statements. The PerformanceInfo table displays PERFORMANCE statement options that are used by the SOLVE statement. The CoforPerfInfo table displays PERFORMANCE statement options that are used by the COFOR statement. If you specify the DETAILS option in the PERFORMANCE statement, then the SOLVE statement also produces the ODS table Timing. The COFOR statement uses the DETAILS option to set default performance options with each loop iteration.

The following statements generate several ODS tables and write each table to a SAS data set:

```
proc optmodel printlevel=2;
   ods output PrintTable=expt ProblemSummary=exps DerivMethods=exdm
              SolverOptions=exso SolutionSummary=exss OptStatistics=exos
              Timing=exti;
   performance details;
   var x{1..2} >= 0;
   min z = 2*x[1] + 3 * x[2] + x[1]**2 + 10*x[2]**2
           + 2.5*x[1]*x[2] + x[1]**3;
   con c1: x[1] - x[2] <= 1;
   con c2: x[1] + 2*x[2] >= 100;
   solve;
   print x;
```

The data set expt contains the PrintTable table and is shown in Figure 5.46. The variable names are COL1 and x.

Figure 5.46 PrintTable ODS Table

PrintTable

Obs	COL1	x
1	1	10.448
2	2	44.776

The data set exps contains the ProblemSummary table and is shown in Figure 5.47. The variable names are Label1, cValue1, and nValue1. The rows describe the instance, and the description depends on the form of the problem. In most solvers, the rows describe the objective function, variables, and constraints. In the network solver, the rows describe the number of nodes, the number of edges, the directedness of the graph, and the type of problem solved over the graph.

Figure 5.47 ProblemSummary ODS Table

ProblemSummary

Obs	Label1	cValue1	nValue1
1	Objective Sense	Minimization	.
2	Objective Function	z	.
3	Objective Type	Nonlinear	.
4			.
5	Number of Variables	2	2.000000
6	Bounded Above	0	0
7	Bounded Below	2	2.000000
8	Bounded Below and Above	0	0
9	Free	0	0
10	Fixed	0	0
11			.
12	Number of Constraints	2	2.000000
13	Linear LE (<=)	1	1.000000
14	Linear EQ (=)	0	0
15	Linear GE (>=)	1	1.000000
16	Linear Range	0	0

The data set exso contains the SolverOptions table and is shown in Figure 5.48. The variable names are Label1, cValue1, nValue1, cValue2, and nValue2. The rows, which depend on the solver called by PROC OPTMODEL, list the values taken by each of the solver options. The presence of an asterisk (*) next to an option indicates that a value has been specified for that option.

Figure 5.48 SolverOptions ODS Table

SolverOptions

Obs	Label1	cValue1	nValue1	cValue2	nValue2
1	ALGORITHM	INTERIORPOINT	.		.
2	FEASTOL	1E-6	0.000001000		.
3	HESSTYPE	FULL	.		.
4	LOGFREQ	1	1.000000		.
5	MAXITER	5000	5000.000000		.
6	MAXTIME	I	I		.
7	OBJLIMIT	1E20	1E20		.
8	OPTTOL	1E-6	0.000001000		.
9	SOLTYPE	1	1.000000		.
10	TIMETYPE	REAL	.		.

The data set exdm contains the DerivMethods table, which displays the methods of derivative computation, and is shown in Figure 5.49. The variable names are Label1, cValue1, and nValue1. The rows, which depend on the derivatives used by the solver, specify the method used to calculate each derivative.

Figure 5.49 DerivMethods ODS Table

DerivMethods

Obs	Label1	cValue1	nValue1
1	Objective Gradient	Analytic Formulas	.
2	Objective Hessian	Analytic Formulas	.

The data set exss contains the SolutionSummary table and is shown in Figure 5.50. The variable names are Label1, cValue1, and nValue1. The rows give an overview of the solution, including the solver chosen, the objective value, and the solution status. Depending on the values returned by the solver, the SolutionSummary table might also include some solution quality values such as optimality error and infeasibility. The values in the SolutionSummary table appear in the _OROPTMODEL_ macro variable; each solver chapter has a section that describes the solver's contribution to this macro variable.

Figure 5.50 SolutionSummary ODS Table

SolutionSummary

Obs	Label1	cValue1	nValue1
1	Solver	NLP	.
2	Algorithm	Interior Point	.
3	Objective Function	z	.
4	Solution Status	Optimal	.
5	Objective Value	22623.347101	22623
6			.
7	Optimality Error	5E-7	0.000000500
8	Infeasibility	0	0
9			.
10	Iterations	5	5.000000
11	Presolve Time	0.00	0.000175
12	Solution Time	0.01	0.006376

The data set exos contains the OptStatistics table, which displays the optimization statistics, and is shown in Figure 5.51. The variable names are Label1, cValue1, and nValue1. The rows, which depend on the solver called by PROC OPTMODEL, describe the amount of time and the function evaluations that are used by the solver and associated processing. Times are displayed in seconds of clock or CPU time according to the value of the TIMETYPE= option that is used by the solver.

Figure 5.51 OptStatistics ODS Table

OptStatistics

Obs	Label1	cValue1	nValue1
1	Function Evaluations	28	28.000000
2	Gradient Evaluations	28	28.000000
3	Hessian Evaluations	6	6.000000
4	Problem Generation Time	0.00	0.000398
5	Code Generation Time	0.01	0.007144
6	Presolve Time	0.00	0.000175
7	Solution Time	0.01	0.006376
8	Total Time	0.07	0.074679

Problem generation is the process of combining the model with the data into a format that solvers can use. This includes computing equation coefficients, but it does not include reading data or evaluating other programming statements. Code generation is compiles code for nonlinear equations in the model and performs other analysis that is needed prior to solver evaluations. The time required for problem generation will be negligible if the model contains only linear equations. The presolve time in this table includes the time used by the PROC OPTMODEL presolver and any presolver that is part of the solver. Solution time is the sum of the times used by the presolvers and the solver. The presolve and solution times also appear in the SolutionSummary table. The OptStatistics table includes a total time, which is the sum of times for problem generation, code generation, solution, and overhead in the SOLVE statement. Overhead includes solver setup, postprocessing, and ODS table output.

The Timing table provides an alternate breakdown of SOLVE statement timing. Times in this table are shown in seconds of clock time. The data set exti, which is shown in Figure 5.52, contains the Timing table data and statistics. The variable names are Label1, cValue1, nValue1, cValue2, and nValue2. The values present depend on the solver and on the context of the SOLVE statement.

Figure 5.52 Timing ODS Table

Timing

Obs	Label1	cValue1	nValue1	cValue2	nValue2
1	Problem Generation	0.0003975585	0.00	0.0053228127	0.53%
2	OPTMODEL Presolver	0.0001746318	0.00	0.0023381017	0.23%
3	Solver Initialization	0.0603277548	0.06	0.8077133928	80.77%
4	Code Generation	0.0071441545	0.01	0.0956513183	9.57%
5	Solver	0.0062017029	0.01	0.0830330661	8.30%
6	Solver Postprocessing	0.0004437537	0.00	0.0059413085	0.59%

Some of the Timing table values have already been described for the OptStatistics table. Solver initialization time is overhead in the SOLVE statement before the solver starts. Solver time includes execution of the solver and its associated preprocessor, if any. Solver postprocessing time is overhead in the SOLVE statement after the solver has completed. Several table values appear only for a SOLVE statement called within a COFOR loop. These values are shown in Table 5.14.

Table 5.14 Solver Timing Values for COFOR Loops

Label	Description
Waiting for Multiple Threads	Time waiting for the required number of threads to become available on the client machine when NTHREADS > 1
Waiting for Grid	Time waiting for the distributed computing environment to become ready to accept a new problem
Sending to Grid	Time to transmit a solver problem description to the distributed computing environment
Receiving from Grid	Time to receive solver results from the distributed computing environment
Waiting after Solver	Time between solver completion and the resumption of the SOLVE statement for processing the results
Grid Overhead	Time required for processing in the distributed computing environment that is not included in the preceding times

Constraints

You can add constraints to a PROC OPTMODEL model. The solver tries to satisfy the specified constraints while minimizing or maximizing the objective.

Constraints in PROC OPTMODEL have names. By using the name, you can examine various attributes of the constraint, such as the dual value that is returned by the solver (see the section "Suffixes" on page 134 for details). A constraint is not allowed to have the same name as any other model component.

PROC OPTMODEL provides a default name if none is supplied by the constraint declaration. The predefined array _ACON_ provides names for otherwise anonymous constraints. The predefined numeric parameter _NACON_ contains the number of such constraints. The constraints are assigned integer indices in sequence, so _ACON_[1] refers to the first unnamed constraint declared, while _ACON_[_NACON_] refers to the newest.

Consider the following example of a simple model that has a constraint:

```
proc optmodel;
   var x, y;
   min r = x**2 + y**2;
   con c: x+y >= 1;
   solve;
   print x y;
```

Without the constraint named c, the solver would find the point $x = y = 0$ that has an objective value of 0. However, the constraint makes this point infeasible. The resulting output is shown in Figure 5.53.

Figure 5.53 Constrained Model Solution

Problem Summary	
Objective Sense	Minimization
Objective Function	r
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	1
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	2

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	r
Solution Status	Optimal
Objective Value	0.4999995397
Primal Infeasibility	2.3014762E-7
Dual Infeasibility	2.3570226E-7
Bound Infeasibility	0
Duality Gap	1.9575231E-7
Complementarity	0
Iterations	3
Presolve Time	0.00
Solution Time	0.01

x	y
0.5	0.5

The solver has found the point where the objective function is minimized in the region $x + y \geq 1$. This is actually on the border of the region: the constraint c is active (see the section "Dual Values" on page 138 for details).

In the preceding example the constraint c had only a lower bound. You can specify constraints that have both upper and lower bounds. For example, replacing the constraint c in the previous example would further restrict the feasible region:

```
con c: 3 >= x+y >= 1;
```

PROC OPTMODEL standardizes constraints to collect the expression terms that depend on variables and to separate the expression terms that are constant. When there is a single equality or inequality operator, the separable constant terms are moved to the right operand while the variable terms are moved to the left operand. For range constraints, the separable constant terms from the middle expression are subtracted from the lower and upper bounds. You can see the standardized constraints with the use of the EXPAND statement in the following example. Consider the following PROC OPTMODEL statements:

```
proc optmodel;
   var x{1..3};
   con b: sum{i in 1..3}(x[i] - i) = 0;
   expand b;
```

These statements produce an optimization problem with the following constraint:

```
(x[1] - 1) + (x[2] - 2) + (x[3] - 3) = 0
```

The EXPAND statement produces the output in Figure 5.54.

Figure 5.54 Expansion of a Standardized Constraint

```
Constraint b: x[1] + x[2] + x[3] = 6
```

Here the i separable constant terms in the operand of the SUM operation were moved to the right-hand side of the constraint. The sum of these i values is 6.

After standardization the constraint expression that contains all the variables is called the *body* of the constraint. You can reference the current value of the body expression by attaching the .body suffix to the constraint name. Similarly, the upper and lower bound expressions can be referenced by using the .ub and .lb suffixes, respectively. (See the section "Suffixes" on page 134 for more information.)

As a result of standardization, the value of a body expression depends on how the corresponding constraint is entered. The following example demonstrates how using equivalent relational syntax can result in different .body values:

```
proc optmodel;
    var x init 1;
    con c1: x**2 <= 5;
    con c2: 5 >= x**2;
    con c3: -x**2 >= -5;
    con c4: -5 <= -x**2;
    expand;
    print c1.body c2.body c3.body c4.body;
```

The EXPAND and PRINT statements produce the output in Figure 5.55.

Figure 5.55 Expansion and Body Values of Standardized Constraints

```
Var x
Constraint c1: x**2 <= 5
Constraint c2: -x**2 >= -5
Constraint c3: -x**2 >= -5
Constraint c4: --x**2 <= 5
```

c1.BODY	c2.BODY	c3.BODY	c4.BODY
1	-1	-1	1

CAUTION: Each constraint has an associated *dual value* (see "Dual Values" on page 138). As a result of standardization, the sign of a dual value depends in some instances on the way in which the corresponding constraint is entered into PROC OPTMODEL. In the case of a minimization objective with one-sided constraint $g(x) \geq L$, avoid entering the constraint as $L \leq g(x)$. For example, the following statements produce a value of 2:

```
proc optmodel;
   var x;
   min o1 = x**2;
   con c1: x >= 1;
   solve;
   print (c1.dual);
```

Replacing the constraint as follows results in a value of –2:

```
   con c1: 1 <= x;
```

In the case of a maximization objective with the one-sided constraint $g(x) \leq U$, avoid entering the constraint as $U \geq g(x)$.

When a constraint has variables on both sides, the sign of the dual value depends on the direction of the inequality. For example, you can enter the following constraint:

```
   con c1: x**5 - y + 8 <= 5*x + y**2;
```

This is a $\leq$ constraint, so c1.dual is nonpositive. If you enter the same constraint as follows, then c1.dual is nonnegative:

```
   con c1: 5*x + y**2 >= x**5 - y + 8;
```

It is also important to note that the signs of the dual values are negated in the case of maximization. The following statements output a value of 2:

```
proc optmodel;
   var x;
   min o1 = x**2;
   con c1: 1 <= x <= 2;
   solve;
   print (c1.dual);
```

Changing the objective function as follows yields the same value of x, but c1.dual now holds the value –2:

```
   max o1 = -x**2;
```

NOTE: A simple bound constraint on a decision variable *x* can be entered either by using a CONSTRAINT declaration or by assigning values to x.lb and x.ub. If you require dual values for simple bound constraints, use the CONSTRAINT declaration.

Constraints that are specified using relational operators can be linear or nonlinear. PROC OPTMODEL determines the type of constraint automatically by examining the form of the body expression. Subexpressions that do not involve variables are treated as constants. Constant subexpressions that are multiplied by or added to linear subexpressions produce new linear subexpressions. For example, constraint A in the following statements is linear:

```
proc optmodel;
   var x{1..3};
   con A: 0.5*(x[1]-x[2]) + x[3] >= 0;
```

Suffixes

Use suffixes with *identifier-expressions* to retrieve and modify various auxiliary values maintained by the solver. The values of the suffixes can come from expressions in the declaration of the name that is suffixed. For example, the following declaration of variable v provides the values of several suffixes of v at the same time:

```
var v >= 0 <= 2 init 1;
```

The values of the suffixes also come from the solver or from values assigned by assignment or READ DATA statements (see an example in the section "Data Set Input/Output" on page 117).

You must use suffixes with names of the appropriate type. For example, the .dual suffix cannot be used with the name of an objective. In particular, local dummy parameter names cannot have suffixes.

Table 5.15 shows the names of the available suffixes.

Table 5.15 Suffix Names

Name Kind	Suffix	Modifiable	Description
any	.name	No	Name text for any non-dummy symbol
Constraint	.active	No	Active status in current problem
Constraint	.block	Yes	Block ID for decomposition
Constraint	.body	No	Current constraint body value
Constraint	.dual	No	Dual value from the solver
Constraint	.label	Yes	Label text for the solver
Constraint	.lb	Yes	Current lower bound
Constraint	.status	Yes	Status information from solver
Constraint	.ub	Yes	Current upper bound
Implicit Variable	.sol	No	Current solution value
Objective	.active	No	Active status in current problem
Objective	.sol	No	Current objective value
Objective	.label	Yes	Label text for the solver
Problem	.active	No	Active status of problem
Problem	.label	Yes	Label text for the solver

Table 5.15 (continued)

Name Kind	Suffix	Modifiable	Description
Variable	.active	No	Active status in current problem
Variable	.direction	Yes	Branching direction for MILP
Variable	.dual	No	Alias for .rc
Variable	.fixed	No	Fixed status
Variable	.label	Yes	Label text for the solver
Variable	.lb	Yes	Lower bound
Variable	.msinit	No	Numeric value at the best starting point reported by multistart solver
Variable	.priority	Yes	Branching priority for MILP and CLP
Variable	.rc	No	Reduced cost (LP) or gradient of Lagrangian function
Variable	.relax	Yes	Relaxation of integrality restriction
Variable	.sol	No	Current variable value
Variable	.sol[i]	Yes	Saved solution value
Variable	.status	Yes	Status information from solver
Variable	.ub	Yes	Upper bound

The .sol suffix for a variable, implicit variable, or objective can be used within a declaration to reference the current value of the symbol. It is treated as a constant in such cases. The value is independent of the current problem. When the OPTMODEL procedure processes a SOLVE statement, the value is fixed at the start of the SOLVE statement. The .sol suffix can be followed by a positive integer solution index to refer to a particular solution that the SOLVE statement returns. See the section "Multiple Solutions" on page 151 for more information about accessing multiple solutions from the solver. Each problem tracks saved solution values separately. Outside of declarations, a variable, implicit variable, or objective name with the .sol suffix and no solution index is equivalent to the unsuffixed name.

The .status suffix reports status information from the solver. Currently, only the LP solver provides status information. The .status suffix takes on the same character values that are found in the _STATUS_ variable of the PRIMALOUT and DUALOUT data sets for the OPTLP procedure, including values set by the IIS= option. See the section "Variable and Constraint Status" on page 273 and the section "Irreducible Infeasible Set" on page 274, both in Chapter 7, "The Linear Programming Solver," for more information. For other solvers, the .status values default to a single blank character.

If you choose to modify the .status suffix for a variable or constraint, the assigned suffix value can be a single character or an empty string. The LP solver rejects invalid status characters. Blank or empty strings are treated as new row or column entries for the purpose of "warm starting" the solver.

The .active suffix reports the current activity status for names in the problem. The value is 1 if the element is active or 0 otherwise. A PROBLEM name is considered active if it is the current problem (that is, it was selected by the most recent USE PROBLEM statement). A constraint is considered active if it is included in the current problem and not dropped. An objective is considered active if it is the selected objective for the current problem. A variable is considered active if it is included in the current problem, independent of the fixed status.

The .fixed suffix reports the fixed status of a variable. The value is 1 if the variable is fixed using the FIX statement for the current problem or 0 otherwise. Variables that are not included in the current problem are treated as unfixed.

The .msinit suffix reports the numeric value of a variable at the best starting point, as reported by the NLP solver when the MULTISTART option is specified. If the solver does not report a best starting point, then the value is missing. The value is tracked independently for each problem to support multiple subproblems. See the section "Multistart" on page 512 in Chapter 10, "The Nonlinear Programming Solver," for more information.

The .block suffix identifies the subproblem for constraints when used with the METHOD=USER option of the decomposition algorithm. The value must be numeric and is initially assigned a missing value. A constraint with a missing value for the .block suffix is part of the master problem. Otherwise constraints belong to the same subproblem if and only if they have the same .block suffix values. See Chapter 15, "The Decomposition Algorithm," for more information.

The .label suffix represents the text passed to the solver to identify a variable, constraint, or objective. Some solvers can display this label in their output. The maximum text length passed to the solver is controlled by the MAXLABLEN= option. The default text is based on the name in the model, abbreviated to fit within MAXLABLEN. For example, a model variable x[1] would be labeled "x[1]". This label text can be reassigned. The .label suffix value is also used to create MPS labels stored in the output data set for the SAVE MPS and SAVE QPS statements.

The .name suffix represents the name of a symbol as a text string. The .name suffix can be used with any declared name except for local dummy parameters. This suffix is primarily useful when applied to problem symbols (see the section "Problem Symbols" on page 152), since the .name suffix returns the name of the referenced symbol, not the problem symbol name. The name text is based on the name in the model, abbreviated to fit in 256 characters.

Suffixed names can be used wherever a parameter name is accepted, provided only the value is required. However, you are not allowed to change the value of certain suffixes. Table 5.15 marks these suffixes as not modifiable. Suffixed names that are used as a target in an assignment or READ DATA statement must be modifiable.

The following statements formulate a trivial linear programming problem. The objective value is unbounded, which is reported after the execution of the SOLVE statement. The PRINT statements illustrate the corresponding default auxiliary values. This is shown in Figure 5.56.

```
proc optmodel;
     var x, y;
     min z = x + y;
     con c: x + 2*y <= 3;
     solve;
     print x.lb x.ub x.status x.sol;
     print y.lb y.ub y.status y.sol;
     print c.lb c.ub c.body c.dual;
```

Figure 5.56 Using a Suffix: Retrieving Auxiliary Values

x.LB	x.UB	x.STATUS	x.SOL
-1.7977E+308	1.7977E+308	I	0

y.LB	y.UB	y.STATUS	y.SOL
-1.7977E+308	1.7977E+308	I	0

c.LB	c.UB	c.BODY	c.DUAL
-1.7977E+308	3	0	.

Next, continue to submit the following statements to change the default bounds and solve again. The output is shown in Figure 5.57.

```
x.lb=0;
y.lb=0;
c.lb=1;
solve;
print x.lb x.ub x.status x.sol;
print y.lb y.ub y.status y.sol;
print c.lb c.ub c.body c.dual;
```

Figure 5.57 Using a Suffix: Modifying Auxiliary Values

x.LB	x.UB	x.STATUS	x.SOL
0	1.7977E+308	L	0

y.LB	y.UB	y.STATUS	y.SOL
0	1.7977E+308	B	0.5

c.LB	c.UB	c.BODY	c.DUAL
1	3	1	0.5

NOTE: Spaces are significant. The form NAME.␣TAG is treated as a SAS format name followed by the tag name, not as a suffixed identifier. The forms NAME.TAG, NAME␣.␣TAG, and NAME␣.TAG (note the location of spaces) are interpreted as suffixed references.

Integer Variable Suffixes

The suffixes .relax, .priority, and .direction are applicable to integer variables.

For an integer variable x, setting x.relax to a nonzero, nonmissing value relaxes the integrality restriction. The value of x.relax is read as either 1 or 0, depending on whether or not integrality is relaxed. This suffix is ignored for noninteger variables.

The value that is contained in x.priority sets the priority of an integer variable x for use with branching in the MILP solver or selection in the CLP solvers. This value can be any nonnegative, nonmissing number. The default value is 0, which indicates default branching priority. Variables with positive .priority values are assigned greater priority than the default. Variables with the highest .priority values are assigned the highest priority. Variables with the same .priority value are assigned the same branching priority.

The value of x.direction assigns a branching direction to an integer variable x. This value should be an integer in the range –1 to 3. A noninteger value in this range is rounded on assignment. The default value is 0. The significance of each integer is found in Table 5.16.

Table 5.16 Branching Directions

Value	Direction
–1	Round down to nearest integer
0	Default
1	Round up to nearest integer
2	Round to nearest integer
3	Round to closest presolved bound

Suppose the solver branches next on an integer variable x whose last LP relaxation solution is 3.3. Suppose also that after passing through the presolver, the lower bound of x is 0 and the upper bound of x is 10. If the value in x.direction is –1 or 2, then the solver sets x to 3 for the next iteration. If the value in x.direction is 1, then the solver sets x to 4. If the value in x.direction is 3, then the solver sets x to 0.

The MPS data set created by the SAVE MPS statement ("SAVE MPS Statement" on page 86) includes a BRANCH section if any nondefault .priority or .direction values have been specified for integer variables.

Dual Values

A dual value is associated with each constraint. To access the dual value of a constraint, use the constraint name followed by the suffix .dual.

For linear programming problems, the dual value associated with a constraint is also known as the dual price (also called the shadow price). The shadow price is usually interpreted economically as the rate at which the optimal value changes with respect to a change in some right-hand side that represents a resource supply or demand requirement.

For nonlinear programming problems, the dual values correspond to the values of the optimal Lagrange multipliers. For more details about duality in nonlinear programming, see Bazaraa, Sherali, and Shetty (1993).

From the dual value associated with the constraint, you can also tell whether the constraint is active or not. A constraint is said to be active (tight at a point) if it holds with equality at that point. It can be informative to identify active constraints at the optimal point and check their corresponding dual values. Relaxing the active constraints might improve the objective value.

Background on Duality in Mathematical Programming

For a minimization problem, there exists an associated problem with the following property: any feasible solution to the associated problem provides a lower bound for the original problem, and conversely any feasible solution to the original problem provides an upper bound for the associated problem. The original and the associated problems are referred to as the primal and the dual problem, respectively. More specifically, consider the primal problem,

$$\begin{array}{lll} \underset{x}{\text{minimize}} & f(x) & \\ \text{subject to} & c_i(x) = 0, & i \in \mathcal{E} \\ & c_i(x) \leq 0, & i \in \mathcal{L} \\ & c_i(x) \geq 0, & i \in \mathcal{G} \end{array}$$

where $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{G}$ denote the sets of equality, $\leq$ inequality, and $\geq$ inequality constraints, respectively. Variables $x \in \mathbb{R}^n$ are called the primal variables. The Lagrangian function of the primal problem is defined as

$$L(x, \lambda, \mu, \nu) = f(x) - \sum_{i \in \mathcal{E}} \lambda_i c_i(x) - \sum_{i \in \mathcal{L}} \mu_i c_i(x) - \sum_{i \in \mathcal{G}} \nu_i c_i(x)$$

where $\lambda_i \in \mathbb{R}$, $\mu_i \leq 0$, and $\nu_i \geq 0$. By convention, the Lagrange multipliers for inequality constraints have to be nonnegative. Hence λ, $-\mu$, and ν correspond to the Lagrange multipliers in the preceding Lagrangian function. It can be seen that the Lagrangian function is a linear combination of the objective function and constraints of the primal problem.

The Lagrangian function plays a fundamental role in nonlinear programming. It is used to define the optimality conditions that characterize a local minimum of the primal problem. It is also used to formulate the dual problem of the preceding primal problem. To this end, consider the following *dual* function:

$$d(\lambda, \mu, \nu) = \inf_x \; L(x, \lambda, \mu, \nu)$$

The dual problem is defined as

$$\begin{array}{ll} \underset{\lambda,\mu,\nu}{\text{maximize}} & d(\lambda, \mu, \nu) \\ \text{subject to} & \mu \leq 0 \\ & \nu \geq 0. \end{array}$$

The variables λ, μ, and ν are called the dual variables. Note that the dual variables associated with the equality constraints (λ) are free, whereas those associated with $\leq$ inequality constraints (μ) have to be nonpositive and those associated with $\geq$ inequality constraints (ν) have to be nonnegative.

The relation between the primal and the dual problems provides a nice connection between the optimal solutions of the problems. Suppose x^* is an optimal solution of the primal problem and $(\lambda^*, \mu^*, \nu^*)$ is an optimal solution of the dual problem. The difference between the objective values of the primal and dual problems, $\delta = f(x^*) - d(\lambda^*, \mu^*, \nu^*) \geq 0$, is called the duality gap. For some restricted class of convex nonlinear programming problems, both the primal and the dual problems have an optimal solution and the optimal objective values are equal—that is, the duality gap $\delta = 0$. In such cases, the optimal values of the dual variables correspond to the optimal Lagrange multipliers of the primal problem with the correct signs.

A maximization problem is treated analogously to a minimization problem. For the maximization problem

$$\begin{array}{ll} \underset{x}{\text{maximize}} & f(x) \\ \text{subject to} & c_i(x) = 0, \quad i \in \mathcal{E} \\ & c_i(x) \leq 0, \quad i \in \mathcal{L} \\ & c_i(x) \geq 0, \quad i \in \mathcal{G}, \end{array}$$

the dual problem is

$$\begin{array}{ll} \underset{\lambda,\mu,\nu}{\text{minimize}} & d(\lambda, \mu, \nu) \\ \text{subject to} & \mu \geq 0 \\ & \nu \leq 0. \end{array}$$

where the dual function is defined as $d(\lambda, \mu, \nu) = \sup_x \; L(x, \lambda, \mu, \nu)$ and the Lagrangian function $L(x, \lambda, \mu, \nu)$ is defined the same as earlier. In this case, λ, μ, and $-\nu$ correspond to the Lagrange multipliers in $L(x, \lambda, \mu, \nu)$.

Minimization Problems

For inequality constraints in minimization problems, a positive optimal dual value indicates that the associated $\geq$ inequality constraint is active at the solution, and a negative optimal dual value indicates that the associated $\leq$ inequality constraint is active at the solution. In PROC OPTMODEL, the optimal dual value for a *range constraint* (a constraint with both upper and lower bounds) is the sum of the dual values associated with the upper and lower inequalities. Since only one of the two inequalities can be active, the sign of the optimal dual value, if nonzero, identifies which one is active.

For equality constraints in minimization problems, the optimal dual values are unrestricted in sign. A positive optimal dual value for an equality constraint implies that, starting close enough to the primal solution, the same optimum could be found if the equality constraint were replaced with a $\geq$ inequality constraint. A negative optimal dual value for an equality constraint implies that the same optimum could be found if the equality constraint were replaced with a $\leq$ inequality constraint.

The following is an example where simple linear programming is considered:

```
proc optmodel;
   var x, y;
   min z = 6*x + 7*y;
   con
      4*x +   y >=  5,
       -x - 3*y <= -4,
        x +   y <=  4;
   solve;
   print x y;
   expand _ACON_ ;
   print _ACON_.dual _ACON_.body;
```

The PRINT statements generate the output shown in Figure 5.58.

Figure 5.58 Dual Values in Minimization Problem: Display

Problem Summary	
Objective Sense	Minimization
Objective Function	z
Objective Type	Linear
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	3
Linear LE (<=)	2
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	6

Figure 5.58 *continued*

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	z
Solution Status	Optimal
Objective Value	13
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	4
Presolve Time	0.00
Solution Time	0.00

x	y
1	1

```
Constraint _ACON_[1]: 4*x + y >= 5
Constraint _ACON_[2]: - x - 3*y <= -4
Constraint _ACON_[3]: x + y <= 4
```

[1]	_ACON_.DUAL	_ACON_.BODY
1	1	5
2	-2	-4
3	0	2

It can be seen that the first and second constraints are active, with dual values 1 and –2. Continue to submit the following statements. Notice how the objective value is changed in Figure 5.59.

```
_ACON_[1].lb = _ACON_[1].lb - 1;
solve;
_ACON_[2].ub = _ACON_[2].ub + 1;
solve;
```

Figure 5.59 Dual Values in Minimization Problem: Interpretation

Problem Summary	
Objective Sense	Minimization
Objective Function	z
Objective Type	Linear
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	3
Linear LE (<=)	2
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	6

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	z
Solution Status	Optimal
Objective Value	12
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	4
Presolve Time	0.00
Solution Time	0.00

Figure 5.59 *continued*

Problem Summary	
Objective Sense	Minimization
Objective Function	z
Objective Type	Linear
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	3
Linear LE (<=)	2
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	6

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	z
Solution Status	Optimal
Objective Value	10
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	4
Presolve Time	0.00
Solution Time	0.00

The change is just as the dual values imply. After the first constraint is relaxed by one unit, the objective value is improved by one unit. For the second constraint, the relaxation and improvement are one unit and two units, respectively.

NOTE: The signs of dual values produced by PROC OPTMODEL depend, in some instances, on the way in which the corresponding constraints are entered. See the section "Constraints" on page 130 for details.

Maximization Problems

For inequality constraints in maximization problems, a positive optimal dual value indicates that the associated $\leq$ inequality constraint is active at the solution, and a negative optimal dual value indicates that the associated $\geq$ inequality constraint is active at the solution. The optimal dual value for a range constraint is the sum of the dual values associated with the upper and lower inequalities. The sign of the optimal dual value identifies which inequality is active.

For equality constraints in maximization problems, the optimal dual values are unrestricted in sign. A positive optimal dual value for an equality constraint implies that, starting close enough to the primal solution, the same optimum could be found if the equality constraint were replaced with a $\leq$ inequality constraint. A negative optimal dual value for an equality constraint implies that the same optimum could be found if the equality constraint were replaced with a $\geq$ inequality constraint.

CAUTION: The signs of dual values produced by PROC OPTMODEL depend, in some instances, on the way in which the corresponding constraints are entered. See the section "Constraints" on page 130 for details.

Reduced Costs

In linear programming problems, each variable has a corresponding reduced cost. To access the reduced cost of a variable, add the suffix .rc or .dual to the variable name. These two suffixes are interchangeable.

The reduced cost of a variable is the rate at which the objective value changes when the value of that variable changes. At optimality, basic variables have a reduced cost of zero; a nonbasic variable with zero reduced cost indicates the existence of multiple optimal solutions.

In nonlinear programming problems, the reduced cost interpretation does not apply. The .dual and .rc variable suffixes represent the gradient of the Lagrangian function, computed using the values returned by the solver.

The following example illustrates the use of the .rc suffix:

```
proc optmodel;
   var x >= 0, y >= 0, z >= 0;
   max cost = 4*x + 3*y - 5*z;
   con
      -x + y + 5*z <= 15,
      3*x - 2*y - z <= 12,
      2*x + 4*y + 2*z <= 16;
   solve;
   print x y z;
   print x.rc y.rc z.rc;
```

The PRINT statements generate the output shown in Figure 5.60.

Figure 5.60 Reduced Cost in Maximization Problem: Display

x	y	z
5	1.5	0

x.RC	y.RC	z.RC
0	0	-6.5

In this example, x and y are basic variables, while z is nonbasic. The reduced cost of z is –6.5, which implies that increasing z from 0 to 1 decreases the optimal value from 24.5 to 18.

Presolver

PROC OPTMODEL includes a simple presolver that processes linear constraints to produce tighter bounds on variables. The presolver can reduce the number of variables and constraints that are presented to the solver. These changes can result in reduced solution times.

Linear constraints that involve only a single variable are converted into variable bounds. The presolver then eliminates redundant linear constraints for which variable bounds force the constraint to always be satisfied. Tightly bounded variables where upper and lower bounds are within the range specified by the VARFUZZ= option (see the section "PROC OPTMODEL Statement" on page 40) are automatically fixed to the average of the bounds. The presolver also eliminates variables that are fixed by the user or by the presolver.

The presolver can infer tighter variable bounds from linear constraints when all variables in the constraint or all but one variable have known bounds. For example, when given the following PROC OPTMODEL declarations, the presolver can determine the bound $y \leq 4$:

```
proc optmodel;
   var x >= 3;
   var y;
   con c: x + y <= 7;
```

The presolver makes multiple passes and rechecks linear constraints after bounds are tightened for the referenced variables. The number of passes is controlled by the PRESOLVER= option. After the passes are finished, the presolver attempts to fix the value of all variables that are not used in the updated objective and constraints. The current value of such a variable is used if the value lies between the variable's upper and lower bounds. Otherwise, the value is adjusted to the nearer bound. The value of an integer variable is rounded before being checked against its bounds.

In some cases the solver might perform better without the presolve transformations, so almost all such transformations are unavailable when the option PRESOLVER=BASIC is specified. However, the presolver still eliminates variables that have values that have been fixed by the FIX statement. To disable the OPTMODEL presolver entirely, use PRESOLVER=NONE. The solver assigns values to any unused, unfixed variables when the option PRESOLVER=NONE is specified.

Model Update

The PROC OPTMODEL modeling language provides several means of modifying a model after it is first specified. You can change the parameter values of the model. You can add new model components. The FIX and UNFIX statements can fix variables to specified values or rescind previously fixed values. The DROP and RESTORE statements can deactivate and reactivate constraints. See also the section "Multiple Subproblems" on page 150 for information on how to maintain multiple models.

To illustrate how these statements work, reconsider the following example from the section "Constraints" on page 130:

```
proc optmodel;
   var x, y;
   min r = x**2 + y**2;
   con c: x+y >= 1;
   solve;
   print x y;
```

As described previously, the solver finds the optimal point x = y = 0.5 with r = 0.5. You can see the effect of the constraint c on the solution by temporarily removing it. You can add the following statements:

```
   drop c;
   solve;
   print x y;
```

This change produces the output in Figure 5.61.

Figure 5.61 Solution with Dropped Constraint

Problem Summary	
Objective Sense	Minimization
Objective Function	r
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	0
Constraint Coefficients	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Figure 5.61 *continued*

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	r
Solution Status	Optimal
Objective Value	0
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Duality Gap	0
Complementarity	0
Iterations	0
Presolve Time	0.00
Solution Time	0.00

x	y
0	0

The optimal point is $x = y = 0$, as expected.

You can restore the constraint c with the RESTORE statement, and you can also investigate the effect of forcing the value of variable x to 0.3. This requires the following statements:

```
restore c;
fix x=0.3;
solve;
print x y c.dual;
```

This produces the output in Figure 5.62.

Figure 5.62 Solution with Fixed Variable

Problem Summary	
Objective Sense	Minimization
Objective Function	r
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	1
Fixed	1
Number of Constraints	1
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	2

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	r
Solution Status	Optimal
Objective Value	0.58
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Duality Gap	0
Complementarity	0
Iterations	0
Presolve Time	0.00
Solution Time	0.01

x	y	c.DUAL
0.3	0.7	1.4

The variable x still has the value that was defined in the FIX statement. The objective value has increased by 0.08 from its constrained optimum 0.5 (see Figure 5.53). The constraint c is active, as confirmed by the positive dual value.

You can return to the original optimization problem by allowing the solver to vary variable x with the UNFIX statement, as follows:

```
unfix x;
solve;
print x y c.dual;
```

This produces the output in Figure 5.63. The model was returned to its original conditions.

Figure 5.63 Solution with Original Model

Problem Summary	
Objective Sense	Minimization
Objective Function	r
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	2
Fixed	0
Number of Constraints	1
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	2

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	r
Solution Status	Optimal
Objective Value	0.4999995397
Primal Infeasibility	2.3014762E-7
Dual Infeasibility	2.3570226E-7
Bound Infeasibility	0
Duality Gap	1.9575231E-7
Complementarity	0
Iterations	3
Presolve Time	0.00
Solution Time	0.01

Figure 5.63 *continued*

x	y	c.DUAL
0.5	0.5	1

Multiple Subproblems

The OPTMODEL procedure enables multiple models to be manipulated easily by using named problems to switch the active model components. Problems keep track of an objective, a set of included variables and constraints, and some status information that is associated with the variables and constraints. Other data, such as parameter values, bounds, and the current value of variables, are shared by all problems.

Problems are declared with the PROBLEM declaration. You can easily switch between problems by using the USE PROBLEM statement. The USE PROBLEM statement makes the specified problem become the current problem. The various statements that generate problem data, such as SOLVE, EXPAND, and SAVE MPS, always operate using the model components included in the current problem.

A problem declaration can specify the problem's initial objective by copying it from the problem named in a FROM clause or by including the objective symbol. This objective can be overridden while the problem is current by declaring a new non-array objective or by executing programming statements that specify a new objective.

Variables can also be included when the problem is current by declaring them or by using the FIX or UNFIX statement. Similarly, constraints can be included when the problem is current by declaring them or by using the RESTORE or DROP statement. There is no way to exclude a variable or constraint item after it has been included in a problem, although the variable or constraint can be fixed or dropped.

Variables that are declared but not included in a problem are treated as constants when a problem is generated, while constraints that are declared but not included are ignored. The solver does not update the values and status for these model components.

A problem also tracks certain other status information that is associated with its included symbols, and this information can be changed without affecting other problems. This information includes the fixed status for variables, and the dropped status for constraints. The following additional data that are tracked by the problem are available through variable and constraint suffixes:

- *var*.STATUS (including IIS status)
- *var*.SOL[*i*] (for each solution *i*)
- *var*.MSINIT
- *var*.RC
- *var*.DUAL (alias of *var*.RC)
- *var*.FIXED
- *con*.STATUS (including IIS status)
- *con*.DUAL
- *con*.BLOCK

The initial problem when OPTMODEL starts is predeclared with the name _START_. This problem can be reinstated again (after other USE PROBLEM statements) with the statement

```
use problem _start_;
```

See "Example 5.5: Multiple Subproblems" on page 174 for example statements that use multiple subproblems.

Multiple Solutions

When a solver finishes, it reports zero or more solutions for the optimization variables of the current problem. Each solution assigns a value to each of the variables in the problem. The SOLVE statement saves these solutions with the current problem. The first reported solution, if any, is also copied into the optimization variables. The number of solutions is available in the predeclared numeric parameter _NSOL_.

NOTE: The network solver does not require optimization variables and has its own conventions for returning multiple solutions.

You can access the solutions that are saved with the problem by adding a solution index to the .sol suffix of the variable name. For example, x.sol[2] would reference the second solution saved for variable x in the current problem. Both the variable name and the suffix can be indexed. For example, assign[3,7].sol[1] refers to the first solution for the array variable element assign[3,7]. The solution index must be an integer in the range 1 to _NSOL_.

The following example illustrates the processing of multiple solutions from the CLP solver:

```
proc optmodel printlevel=0;
   var x{1..2} integer >= 1 <= 3;
   con c: alldiff(x);
   solve with clp / allsolns;
   print _NSOL_;
   print {j in 1..2, i in 1.._NSOL_} x[j].sol[i];
   create data solout from [sol]={i in 1.._NSOL_}
       {j in 1..2} <col("x"||j)=x[j].sol[i]> ;
```

This program produces the output in Figure 5.64. It also creates a data set, solout, which has each solution in a separate observation.

Figure 5.64 Processing Multiple Solutions

NSOL
6

	x.SOL					
	1	2	3	4	5	6
1	1	1	2	2	3	3
2	2	3	1	3	1	2

Problem Symbols

The OPTMODEL procedure declares a number of symbols that are aliases for model components in the current problem. These symbols allow the model components to be accessed uniformly. These symbols are described in Table 5.17.

Table 5.17 Problem Symbols

Symbol	Indexing	Description
NVAR		Number of variables
VAR	{1.._NVAR_}	Variable array
NCON		Number of constraints
CON	{1.._NCON_}	Constraint array
_S_NVAR_		Number of presolved variables
_S_VAR_	{1.._S_VAR_}	Presolved variable array
_S_NCON_		Number of presolved constraints
_S_CON_	{1.._S_CON_}	Presolved constraint array
OBJ		Current objective
PROBLEM		Current problem

If the table specifies indexing, then the corresponding symbol is accessed as an array. For example, if the problem includes two variables, x and y, then the value of _NVAR_ is 2 and the current variable values can be accessed as _var_[1] and _var_[2]. The problem variables prefixed with _S are restricted to model components in the problem after processing by the OPTMODEL presolver.

The following statements define a simple linear programming model and then use the problem symbols to print out some of the problem results. The .name suffix is used in the PRINT statements to display the actual variable and constraint names. Any of the suffixes that apply to a model component can be applied to the corresponding generic symbol.

```
proc optmodel printlevel=0;
   var x1 >= 0, x2 >= 0, x3 >= 0, x4 >= 0, x5 >= 0;

   minimize z = x1 + x2 + x3 + x4;

   con a1: x1 + x2 + x3          <= 4;
   con a2:               x4 + x5 <= 6;
   con a3: x1 +          x4      >= 5;
   con a4:      x2 +          x5 >= 2;
   con a5:           x3          >= 3;

   solve with lp;

   print _var_.name _var_ _var_.rc _var_.status;
   print _con_.name _con_.lb _con_.body _con_.ub _con_.dual _con_.status;
```

The PRINT statement output is shown in Figure 5.65.

Figure 5.65 Problem Symbol Output

[1]	_VAR_.NAME	_VAR_	_VAR_.RC	_VAR_.STATUS
1	x1	1	0	B
2	x2	0	1	L
3	x3	3	0	B
4	x4	4	0	B
5	x5	2	0	B

[1]	_CON_.NAME	_CON_.LB	_CON_.BODY	_CON_.UB	_CON_.DUAL	_CON_.STATUS
1	a1	-1.7977E308	4	4.0000E+00	0	L
2	a2	-1.7977E308	6	6.0000E+00	0	B
3	a3	5	5	1.7977E+308	1	U
4	a4	2	2	1.7977E+308	0	U
5	a5	3	3	1.7977E+308	1	U

OPTMODEL Options

All PROC OPTMODEL options can be specified in the PROC statement (see the section "PROC OPTMODEL Statement" on page 40 for more information). However, it is sometimes necessary to change options after other PROC OPTMODEL statements have been executed. For example, if an optimization technique had trouble with convergence, then it might be useful to vary the PRESOLVER= option value. This can be done with the RESET OPTIONS statement.

The RESET OPTIONS statement accepts options in the same form used by the PROC OPTMODEL statement. The RESET OPTIONS statement is also able to reset option values and to change options programmatically. For example, the following statements print the value of parameter n at various precisions:

```
proc optmodel;
   number n = 1/7;
   for {i in 1..9 by 4}
   do;
      reset options pdigits=(i);
      print n;
   end;
   reset options pdigits; /* reset to default */
```

The output generated is shown in Figure 5.66. The RESET OPTIONS statement in the DO loop sets the PDIGITS option to the value of i. The final RESET OPTIONS statement restores the default option value, because the value was omitted.

Figure 5.66 Changing the PDIGITS Option Value

n
0.1

n
0.14286

n
0.142857143

Automatic Differentiation

PROC OPTMODEL automatically generates statements to evaluate the derivatives for most objective expressions and nonlinear constraints. PROC OPTMODEL generates analytic derivatives for objective and constraint expressions written in terms of the procedure's mathematical operators and most standard SAS library functions.

NOTE: Some functions, such as ABS, FLOOR, and SIGN, and some operators, such as IF-THEN, <> (maximum operator), and >< (minimum operator), must be used carefully in modeling expressions because functions that include such components are not continuously differentiable or even continuous.

Expressions that reference user-defined functions or some SAS library functions might require numerical approximation of derivatives. PROC OPTMODEL uses either forward-difference approximation or central-difference approximation as specified by the FD= option (see the section "PROC OPTMODEL Statement" on page 40).

NOTE: The numerical gradient approximations are significantly slower than automatically generated derivatives when the number of optimization variables is large.

Forward-Difference Approximations

The FD=FORWARD option requests the use of forward-difference derivative approximations. For a function f of n variables, the first-order derivatives are approximated by

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + e_i h_i) - f(x)}{h_i}$$

Notice that up to n additional function calls are needed here. The step lengths $h_i, i = 1, \ldots, n$, are based on the assumed function precision, *DIGITS*:

$$h_i = 10^{-DIGITS/2}(1 + |x_i|)$$

You can use the FDIGITS= option to specify the function precision, *DIGITS*, for the objective function. For constraints, use the CDIGITS= option.

The second-order derivatives are approximated by using up to $n(n + 3)/2$ extra function calls (Dennis and Schnabel 1983, pp. 80, 104):

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{f(x + h_i e_i) - 2f(x) + f(x - h_i e_i)}{h_i^2}$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i) - f(x + h_j e_j) + f(x)}{h_i h_j}$$

Notice that the diagonal of the Hessian uses a central-difference approximation (Abramowitz and Stegun 1972, p. 884). The step lengths are

$$h_i = 10^{-DIGITS/3}(1 + |x_i|)$$

Central-Difference Approximations

The FD=CENTRAL option requests the use of central-difference derivative approximations. Generally, central-difference approximations are more accurate than forward-difference approximations, but they require more function evaluations. For a function f of n variables, the first-order derivatives are approximated by

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + e_i h_i) - f(x - e_i h_i)}{2h_i}$$

Notice that up to $2n$ additional function calls are needed here. The step lengths h_i, $i = 1, \ldots, n$, are based on the assumed function precision, *DIGITS*:

$$h_i = 10^{-DIGITS/3}(1 + |x_i|)$$

You can use the FDIGITS= option to specify the function precision, *DIGITS*, for the objective function. For constraints, use the CDIGITS= option.

The second-order derivatives are approximated by using up to $2n(n + 1)$ extra function calls (Abramowitz and Stegun 1972, p. 884):

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{-f(x + 2h_i e_i) + 16f(x + h_i e_i) - 30f(x) + 16f(x - h_i e_i) - f(x - 2h_i e_i)}{12h_i^2}$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i - h_j e_j) - f(x - h_i e_i + h_j e_j) + f(x - h_i e_i - h_j e_j)}{4h_i h_j}$$

The step lengths are

$$h_i = 10^{-DIGITS/3}(1 + |x_i|)$$

Conversions

Numeric values are implicitly converted to strings when needed for function arguments or operands to the string concatenation operator (||). A warning message is generated when the conversion is applied to a function argument. The conversion uses BEST12. format. Unlike the DATA step, the conversion trims blanks.

Implicit conversion of strings to numbers is not permitted. Use the INPUT function to explicitly perform such conversions.

FCMP Routines

The OPTMODEL procedure can call functions and subroutines that are compiled by the FCMP procedure. You can use FCMP functions wherever a function expression is allowed in PROC OPTMODEL. Use the CALL statement to call FCMP subroutines. The following example defines a function in the FCMP procedure and calls it within PROC OPTMODEL:

```
proc fcmp outlib=work.funcs.test;
   /* arithmetic geometric mean */
   function agm(a0, b0);
      a=a0; b = b0;
      if a<=0 or b<=0 then return(0);
      do until( a - b < a/1e12 );
         a1 = 0.5*a + 0.5*b;
         b1 = sqrt(a*b);
         a = a1; b = b1;
      end;
      return(a);
   endsub;
run;

/* libraries must be specified with the CMPLIB option */
option cmplib=work.funcs;

proc optmodel;
   print (agm(1,2));

   /* find x where agm(1,x) == 23 */
   var x init 1;
   num c = 23;
   min z=(agm(1,x)-c)^2;
   solve;
   print x;
```

FCMP subroutines can return data by updating OPTMODEL numeric and string parameters, which are passed as arguments in a CALL statement. These arguments are declared using the OUTARGS statement in the PROC FCMP subroutine definition. The OPTMODEL argument must be specified with an identifier expression. The following code shows a simple example of output arguments. The maximum length of output strings from OUTARGS arguments is restricted to the argument length before the call, as described in the section "CALL Statement" on page 54.

```
proc fcmp outlib=work.funcs.test;
   subroutine do_sqr(x, sq, text $);
      outargs sq, text;
      sq = x*x;
      text = 'This is an example of output arguments';
   endsub;
run;

option cmplib=work.funcs;

proc optmodel;
```

```
string s init repeat(' ', 79); /* reserve 80 bytes */
number n;
call do_sqr(7, n, s);
print s n;
```

This code produces the output in Figure 5.67.

Figure 5.67 FCMP Output Arguments

s	n
This is an example of output arguments	49

You can pass OPTMODEL arrays to FCMP functions and subroutines that accept matrix arguments. The array must match the type and dimensions of the FCMP argument declaration. The argument in the OPTMODEL CALL statement must be specified using the following syntax:

array-name [**.** *suffix*]

The following code passes a constant matrix to an FCMP function. The array coeff contains the coefficients of a polynomial, which in this case defines a simple quadratic formula, $x^2 - 2x + 1$.

```
proc fcmp outlib=work.funcs.test;
   function evalpoly(x, coeff[*]);
      z = 0;
      do i = dim1(coeff) to 1 by -1;
         z = z * x + coeff[i];
      end;
      return (z);
   endsub;
run;

option cmplib=work.funcs;

proc optmodel;
   num coeff{1..3} = [1, -2, 1];
   var x;
   min z=evalpoly(x, coeff);
   solve;
   print x;
```

An array that is used as a matrix argument must be indexed like an FCMP matrix. In other words, the array index set must be specified as the crossproduct of one or more range expressions (such as 1..N) where the lower bound and step size are 1. Set parameters that are used for indexing must contain a crossproduct of ranges, but the element order is not important. The following code shows some examples of suitable and unsuitable array declarations:

```
proc fcmp outlib=work.funcs.test;
   subroutine mattest(x[*]);
      put x[1]=;
   endsub;
   subroutine mattest2(x[*,*]);
      put x[1,1]=;
   endsub;
run;
```

```
option cmplib=work.funcs;

proc optmodel;
   /* the following arrays can be used as matrices */
   num N init 3;
   num mat1{1..N} init 0;
   call mattest(mat1);     /* OK */
   set S1 = 1..5;
   num mat2{S1} init 0;
   call mattest(mat2);     /* OK */
   set S2 = {S1,S1};
   num mat3{S2} init 0;
   call mattest2(mat3);    /* OK */
   num mat4{S1 cross S1} init 0;
   call mattest2(mat4);    /* OK */
   num L init 1;
   num mat5{L..N} init 0;
   call mattest(mat5);     /* OK */
   set S3 init S1;
   num mat6{S3} init 0;
   call mattest(mat6);     /* OK */

   /* some errors are detected at execution time */
   S3 = 2..5;
   call mattest(mat6);     /* ERROR: lower bound not 1 */
   S3 = {1, 3, 4, 5};
   call mattest(mat6);     /* ERROR: missing index 2 */
   L = 0;
   call mattest(mat5);     /* ERROR: lower bound not 1 */

   /* the following arrays cannot be used as matrices */
   num arr1{1..10 by 3};   /* step size is not 1 */
   call mattest(arr1);     /* ERROR */
   num arr2{i in 1..N, j in 1..N: j >= i}; /* selection expression used */
   call mattest2(arr2);    /* ERROR */
   num arr3{i in 1..N, j in 1..i};         /* index dependency on 'i' */
   call mattest2(arr3);    /* ERROR */
```

Not all PROC FCMP functionality is compatible with PROC OPTMODEL; in particular, the following FCMP functions are not supported and should not be called within your FCMP function definitions: READ_ARRAY, WRITE_ARRAY, RUN_MACRO, and RUN_SASFILE. In many cases, OPTMODEL capabilities can replace these functions. Matrix arguments can be used in place of the READ_ARRAY function by using the READ DATA statement to load the matrix in PROC OPTMODEL. Similarly, you can replace the WRITE_ARRAY function in an FCMP subroutine by copying the matrix to an output argument and using the OPTMODEL procedure to write the matrix. You can use the SUBMIT statement in place of the RUN_MACRO and RUN_SASFILE functions.

The SAS CMPLIB= system option specifies where to look for previously compiled functions and subroutines. For more information about the CMPLIB= system option, see *SAS System Options: Reference*. FCMP functions can be used in distributed mode with the NLP multistart solver. The needed PROC FCMP compiled routines are automatically packaged and distributed. For more information about the multistart solver, see Chapter 10, "The Nonlinear Programming Solver," in this book.

NOTE: Distributed mode requires SAS High-Performance Optimization.

PROC OPTMODEL uses derivatives values that are provided by FCMP when they are available. FCMP cannot provide derivatives with respect to array arguments, so PROC OPTMODEL must use finite differences to compute these derivatives. Also, if the CMPOPT= SAS system option specifies the FUNCDIFFERENCING value, then PROC OPTMODEL uses its own finite differencing for FCMP functions.

More on Index Sets

Dummy parameters behave like parameters but are assigned values only when an index set is evaluated. You can reference the declared dummy parameters from index set expressions that follow the index set item. You can also reference the dummy parameters in the expression or statement controlled by the index set. As the members of the set expression of an index set item are enumerated, the element values of the members are assigned to the local dummy parameters.

The number of names in a dummy parameter declaration must match the element length of the corresponding set expression in the index set item. A single name is allowed when the set member type is scalar (numeric or string). If the set members are tuples that have $n > 1$ elements, then n names are required between the angle brackets (< >) that precede the IN keyword.

Multiple index set items in an index set are nominally processed in a left-to-right order. That is, a set expression from an index set item is evaluated as though the index set items that precede it have already been evaluated. The left-hand index set items can assign values to local dummy parameters that are used by the set expressions that follow them. After each member from the set expression is enumerated, any index set items to the right are reevaluated as needed. The actual order in which index set items are evaluated can vary, if necessary, to allow more efficient enumeration. PROC OPTMODEL generates the same set of values in any case, although possibly in a different order than strict left-to-right evaluation.

You can view the element combinations that are generated from an index set as tuples. This is especially true for index set expressions (see the section "Index Set Expression" on page 107). However, in most cases no tuple set is actually formed, and the element values are assigned only to local dummy parameters.

You can specify a selection expression following a colon (:). The index set generates only those combinations of values for which the selection expression is true. For example, the following statements produce a set of upper triangular indices:

```
proc optmodel;
   put (setof {i in 1..3, j in 1..3 : j >= i} <i, j>);
```

These statements produce the output in Figure 5.68.

Figure 5.68 Upper Triangular Index Set

```
{<1,1>,<1,2>,<1,3>,<2,2>,<2,3>,<3,3>}
```

You can use the left-to-right evaluation of index set items to express the previous set more compactly. The following statements produce the same output as the previous statements:

```
proc optmodel;
   put ({i in 1..3, i..3});
```

In this example, the first time the second index set item is evaluated, the value of the dummy parameter i is 1, so the item produces the set {1,2,3}. At the second evaluation the value of i is 2, so the second item produces the set {2,3}. At the final evaluation the value of i is 3, so the second item produces the set {3}.

In many cases it is useful to combine the SLICE operator with index sets. A special form of index set item uses the SLICE operator implicitly. Normally an index set item that is applied to a set of tuples of length greater than one must be of the form

< *name-1* [**,** . . . *name-n*] **>** **IN** *set-expression*

In the special form, one or more of the name elements are replaced by expressions. The expressions select tuple elements by using the SLICE operator. An expression that consists of a single name must be enclosed in parentheses to distinguish it from a dummy parameter. The remaining names are the dummy parameters for the index set item that is applied to the SLICE result. The following example demonstrates the use of implicit set slicing:

```
proc optmodel;
   number N = 3;
   set<num,str> S = {<1,'a'>,<2,'b'>,<3,'a'>,<4,'b'>};
   put ({i in 1..N, <(i),j> in S});
   put ({i in 1..N, j in slice(<i,*>, S)});
```

The two PUT statements in this example are equivalent.

Threaded and Distributed Processing

The OPTMODEL procedure can take advantage of the multiple CPUs that are available in many computers. PROC OPTMODEL automatically uses multithreaded execution to divide problem generation among the multiple CPUs of the computer that is running the procedure. Hessian and Jacobian matrix evaluation is automatically parallelized across threads of execution on multiple CPUs. The COFOR statement enables solvers to concurrently execute in background threads on multiple CPUs, overlapping with PROC OPTMODEL statement processing. Parallel execution can decrease the amount of clock time required to perform a task, although the total CPU time required might increase.

If you use the PERFORMANCE statement and specify an NTHREADS= option, and the statement does not request distributed computing, then threading in the OPTMODEL procedure is controlled by the statement's NTHREADS= option. Otherwise, threading in the OPTMODEL procedure is controlled by the following SAS system options:

CPUCOUNT=*number* | **ACTUAL**
specifies the maximum number of CPUs that can be used.

THREADS | **NOTHREADS**
enables or disables the use of threading.

Good performance is usually obtained with the default option settings (THREADS and CPUCOUNT=ACTUAL). See the option descriptions in *SAS System Options: Reference* for more details.

The PERFORMANCE statement and the SAS system options set the maximum number of threads. The number of threads that PROC OPTMODEL actually uses depends on the characteristics of the problem that is being solved. In particular, threading is not used when the problem is simple enough that threading offers no advantage.

The COFOR statement and certain solver features can use distributed computing when the PERFORMANCE statement specifies a distributed computing environment. Distributed computing provides the most benefit when a computational process can be structured so that it includes a large number of nontrivial subprocesses that can be executed independently.

NOTE: Distributed computing mode requires SAS High-Performance Optimization.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure creates a macro variable named _OROPTMODEL_. You can inspect the execution of the most recently invoked solver from the value of the macro variable. The macro variable is defined at the start of the procedure and updated after each SOLVE statement is executed. The OPTMODEL procedure also updates the macro variable when an error is detected.

The _OROPTMODEL_ value is a string that consists of several "KEYWORD=*value*" items in sequence, separated by blanks; for example:

```
STATUS=OK ALGORITHM=DS SOLUTION_STATUS=OPTIMAL OBJECTIVE=119302.04331
PRIMAL_INFEASIBILITY=3.552714E-13 DUAL_INFEASIBILITY=2.273737E-13
BOUND_INFEASIBILITY=0 ITERATIONS=82 PRESOLVE_TIME=0.02 SOLUTION_TIME=0.05
```

The information contained in _OROPTMODEL_ varies according to which solver was last called. For lists of keywords and possible values, see the individual solver chapters.

If a value has not been computed, then the corresponding element is not included in the value of the macro variable. When PROC OPTMODEL starts, for example, the macro variable value is set to "STATUS=OK" because no SOLVE statement has been executed. If the STATUS= indicates an error, then the other values from the solver might not be available, depending on when the error occurred.

Solver Status Parameters

In addition to creating the macro variable _OROPTMODEL_, the OPTMODEL procedure creates several predeclared parameters to provide simple access to solver status values. These parameters are declared as follows:

```
string _STATUS_;
string _SOLUTION_STATUS_;
set<string> _OROPTMODEL_STR_KEYS_;
set<string> _OROPTMODEL_NUM_KEYS_;
string _OROPTMODEL_STR_{_OROPTMODEL_STR_KEYS_};
number _OROPTMODEL_NUM_{_OROPTMODEL_NUM_KEYS_};
```

The value of _STATUS_ is equal to the STATUS= component of the _OROPTMODEL_ macro variable. The value of _STATUS_ is initially "OK". The value is updated during the SOLVE statement and after statement execution errors.

The value of _SOLUTION_STATUS_ is equal to the SOLUTION_STATUS= component of the _OROPTMODEL_ macro variable. The value is initially an empty string. The value is updated during the SOLVE statement.

You can use the remaining status parameters to access all the components of the _OROPTMODEL_ macro variable. The following statements demonstrate these parameters:

```
proc optmodel printlevel=0;
   var x init 1 >= 0.001;
   min z=sin(x)/x;
   solve;
   for {k in /STATUS SOLUTION_STATUS ALGORITHM/}
      put _OROPTMODEL_STR_[k]=;
   for {k in /OBJECTIVE ITERATIONS/}
      put _OROPTMODEL_NUM_[k]=;
```

These statements produce the output in Figure 5.69.

Figure 5.69 Solver Status Parameters

```
_OROPTMODEL_STR_[STATUS]=OK
_OROPTMODEL_STR_[SOLUTION_STATUS]=OPTIMAL
_OROPTMODEL_STR_[ALGORITHM]=IP
_OROPTMODEL_NUM_[OBJECTIVE]=-0.217233628
_OROPTMODEL_NUM_[ITERATIONS]=3
```

The _OROPTMODEL_STR_ array contains the same character component values that are found in the _OROPTMODEL_ macro variable. Specify the component name as the array index. For example, the indices “STATUS” and “SOLUTION_STATUS” select array elements that hold the STATUS= and SOLUTION_STATUS= component values, respectively. The set _OROPTMODEL_STR_KEYS_ contains the component indices that you can use with _OROPTMODEL_STR_. The OPTMODEL procedure updates the _OROPTMODEL_STR_ and _OROPTMODEL_STR_KEYS_ parameters during the execution of the SOLVE statement and after any execution errors occur.

The _OROPTMODEL_NUM_ array contains the numeric component values that are displayed in the _OROPTMODEL_ macro variable. For example, the index “OBJECTIVE” selects the array element that holds the objective value when a solution is available. The set _OROPTMODEL_NUM_KEYS_ contains the component indices that you can use with _OROPTMODEL_NUM_. The OPTMODEL procedure updates the _OROPTMODEL_NUM_ and _OROPTMODEL_NUM_KEYS_ parameters during the execution of the SOLVE statement and after any execution errors occur.

Macro and Statement Evaluation Order

PROC OPTMODEL reads a complete statement, such as a DO statement, before executing any code in it. But macro language statements are processed as the code is read. So you must be careful when using the _OROPTMODEL_ macro variable in code that involves SOLVE statements nested in loops or DO statements. The following statements demonstrate one example of this behavior:

```
proc optmodel;
   var x, y;
   min z=x**2 + (x*y-1)**2;
   for {n in 1..3} do;
      fix x=n;
      solve;
      %put Line 1 &_OROPTMODEL_;
      put 'Line 2 ' (symget("_OROPTMODEL_"));
   end;
quit;
```

In the preceding statements the %PUT statement is executed once, before any SOLVE statements are executed. It displays PROC OPTMODEL's initial setting of the macro variable. But the PUT statement is executed after each SOLVE statement and indicates the expected solution status.

Rewriting PROC NLP Models for PROC OPTMODEL

This section describes techniques for converting PROC NLP models to PROC OPTMODEL models. Example 5.8 also demonstrates how to rewrite a PROC NLP model for use with PROC OPTMODEL.

To illustrate the basics, consider the following first version of the PROC NLP model for the example "Simple Pooling Problem" in Chapter 6, "The NLP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*):

```
proc nlp all;
   parms amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy pools = 1;
   bounds 0 <= amountx amounty amounta amountb amountc,
               amountx <= 100,
               amounty <= 200,
          0 <= pooltox pooltoy ctox ctoy,
          1 <= pools <= 3;
   lincon amounta + amountb = pooltox + pooltoy,
          pooltox + ctox = amountx,
          pooltoy + ctoy = amounty,
          ctox + ctoy    = amountc;
   nlincon nlc1-nlc2 >= 0.,
           nlc3 = 0.;
   max f;
   costa = 6; costb = 16; costc = 10;
   costx = 9; costy = 15;
   f = costx * amountx + costy * amounty
       - costa * amounta - costb * amountb - costc * amountc;
   nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
   nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
   nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
run;
```

These statements define a model that has bounds, linear constraints, nonlinear constraints, and a simple objective function. The following statements are a straightforward conversion of the PROC NLP statements to PROC OPTMODEL form:

```
proc optmodel;
   var amountx init 1 >= 0 <= 100,
       amounty init 1 >= 0 <= 200;
   var amounta init 1 >= 0,
       amountb init 1 >= 0,
       amountc init 1 >= 0;
   var pooltox init 1 >= 0,
       pooltoy init 1 >= 0;
   var ctox init 1 >= 0,
       ctoy init 1 >= 0;
   var pools init 1 >=1 <= 3;
   con amounta + amountb = pooltox + pooltoy,
       pooltox + ctox = amountx,
       pooltoy + ctoy = amounty,
       ctox + ctoy    = amountc;
   number costa, costb, costc, costx, costy;
   costa = 6; costb = 16; costc = 10;
   costx = 9; costy = 15;
   max f = costx * amountx + costy * amounty
         - costa * amounta - costb * amountb - costc * amountc;
   con nlc1: 2.5 * amountx - pools * pooltox - 2. * ctox >= 0,
       nlc2: 1.5 * amounty - pools * pooltoy - 2. * ctoy >= 0,
       nlc3: 3 * amounta + amountb - pools * (amounta + amountb)
             = 0;
   solve;
   print amountx amounty amounta amountb amountc
         pooltox pooltoy ctox ctoy pools;
```

The PROC OPTMODEL variable declarations are split into individual declarations because PROC OPTMODEL does not permit name lists in its declarations. In the OPTMODEL procedure, you specify variable bounds as part of the variable declaration instead of in a separate BOUNDS statement. The PROC NLP statements are as follows:

```
parms amountx amounty amounta amountb amountc
      pooltox pooltoy ctox ctoy pools = 1;
bounds 0 <= amountx amounty amounta amountb amountc,
           amountx <= 100,
           amounty <= 200,
       0 <= pooltox pooltoy ctox ctoy,
       1 <= pools <= 3;
```

The following PROC OPTMODEL statements are equivalent to the preceding PROC NLP statements:

```
var amountx init 1 >= 0 <= 100,
    amounty init 1 >= 0 <= 200;
var amounta init 1 >= 0,
    amountb init 1 >= 0,
    amountc init 1 >= 0;
var pooltox init 1 >= 0,
    pooltoy init 1 >= 0;
var ctox init 1 >= 0,
    ctoy init 1 >= 0;
var pools init 1 >= 1 <= 3;
```

The linear constraints are declared in the PROC NLP model by using the following statement:

```
lincon amounta + amountb = pooltox + pooltoy,
       pooltox + ctox = amountx,
       pooltoy + ctoy = amounty,
       ctox + ctoy    = amountc;
```

The following linear constraint declarations in the PROC OPTMODEL model are quite similar to the PROC NLP LINCON declarations:

```
con amounta + amountb = pooltox + pooltoy,
    pooltox + ctox = amountx,
    pooltoy + ctoy = amounty,
    ctox + ctoy    = amountc;
```

But PROC OPTMODEL provides much more flexibility in defining linear constraints. For example, a coefficient can be a named parameter or any other expression that evaluates to a constant.

The cost parameters are declared explicitly in the PROC OPTMODEL model. Unlike the DATA step or the NLP procedure, PROC OPTMODEL requires names to be declared before they are used. There are multiple ways to set the values of these parameters. The preceding example uses assignments. You could make the values part of the declaration by using the INIT *expression* clause or the = *expression* clause. You could also read the values from a data set by using the READ DATA statement.

In the original PROC NLP statements, the assignment to a parameter such as costa occurs every time the objective function is evaluated. However, the assignment occurs just once in the PROC OPTMODEL statements, when the assignment statement is processed. This works because the values are constant. But the PROC OPTMODEL statements permit the parameters to be reassigned later so that you can interactively modify the model.

The following statements define the objective f in the PROC NLP model:

```
max f;
. . .
f = costx * amountx + costy * amounty
    - costa * amounta - costb * amountb - costc * amountc;
```

The PROC OPTMODEL version of the objective is defined by using the same expression text, as follows:

```
max f = costx * amountx + costy * amounty
      - costa * amounta - costb * amountb - costc * amountc;
```

But the MAX statement and the assignment to the name f in the PROC NLP statements are combined in PROC OPTMODEL. There are advantages and disadvantages to this approach. The PROC OPTMODEL formulation is much closer to the mathematical formulation of the model. However, if multiple intermediate variables are used to structure the objective, then multiple IMPVAR declarations are required.

In the PROC NLP model, the nonlinear constraints use the following syntax:

```
nlincon nlc1-nlc2 >= 0.,
        nlc3 = 0.;
. . .
nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
```

In the PROC OPTMODEL model, the equivalent statements are as follows:

```
con nlc1: 2.5 * amountx - pools * pooltox - 2. * ctox >= 0,
    nlc2: 1.5 * amounty - pools * pooltoy - 2. * ctoy >= 0,
    nlc3: 3 * amounta + amountb - pools * (amounta + amountb)
          = 0;
```

The nonlinear constraints in PROC OPTMODEL use the same syntax as linear constraints. In fact, if the variable pools were declared as a parameter, then all the preceding constraints would be linear. The nonlinear constraint in PROC OPTMODEL combines the NLINCON statement of PROC NLP with the assignment in the PROC NLP statements. Objective names can be used in nonlinear constraint expressions to structure the formula as they are in objective expressions,

The PROC OPTMODEL model does not use a RUN statement to invoke the solver. Instead the solver is invoked interactively by the SOLVE statement in PROC OPTMODEL. By default, the OPTMODEL procedure prints much less information about the optimization process. Generally this information consists of messages from the solver (such as the termination reason) and a short status display. The PROC OPTMODEL statements add a PRINT statement in order to display the variable estimates from the solver.

Examples: OPTMODEL Procedure

Example 5.1: Matrix Square Root

This example demonstrates the use of PROC OPTMODEL array parameters and variables. The following statements create a randomized positive definite symmetric matrix and define an optimization model to find the matrix square root of the generated matrix:

```
proc optmodel;
   number n = 5;  /* size of matrix */
   /* random original array */
   number A{1..n, 1..n} = 10 - 20*rand('UNIFORM');
   /* compute upper triangle of the
    * symmetric matrix A*transpose(A) */
   /* should be positive def unless A is singular */
   number P{i in 1..n, j in i..n};
   for {i in 1..n, j in i..n}
       P[i,j] = sum{k in 1..n} A[i,k]*A[j,k];
   /* coefficients of square root array
    * (upper triangle of symmetric matrix) */
   var q{i in 1..n, i..n};
   /* The default initial value q[i,j]=0 is
    * a local minimum of the objective,
    * so you must move it away from that point. */
   q[1,1] = 1;
   /* minimize difference of square of q from P */
   min r = sum{i in 1..n, j in i..n}
           (   sum{k in 1..i} q[k,i]*q[k,j]
             + sum{k in i+1..j} q[i,k]*q[k,j]
             + sum{k in j+1..n} q[i,k]*q[j,k]
             - P[i,j] )**2;
   solve;
   print q;
```

These statements define a random array **A** of size $n \times n$. The product **P** is defined as the matrix product AA^T. The product is symmetric, so the declaration of the parameter P gives it upper triangular indexing. The matrix represented by P should be positive definite unless **A** is singular. But singularity is unlikely because of the random generation of **A**. If **P** is positive definite, then it has a well-defined square root, **Q**, such that $P = QQ^T$.

The objective r simply minimizes the sum of squares of the coefficients as

$$r = \sum_{1 \le i \le j \le n} R_{i,j}^2$$

where $R = QQ^T - P$. (This technique for computing matrix square roots is intended only for the demonstration of PROC OPTMODEL capabilities. Better methods exist.)

Output 5.1.1 shows part of the output from running these statements. The values that are actually displayed depend on the random numbers generated.

Output 5.1.1 Matrix Square Root Results

	q				
	1	2	3	4	5
1	-0.10556	-7.03961	8.64638	1.89284	-8.28542
2		5.23609	0.64462	-6.63339	6.71074
3			-1.61894	-7.31865	1.14428
4				3.76627	0.32063
5					4.93412

Example 5.2: Reading From and Creating a Data Set

This example demonstrates how to use the READ DATA statement to read parameters from a SAS data set. The objective is the Bard function, which is the following least squares problem with $I = \{1, 2, \ldots, 15\}$:

$$f(x) = \frac{1}{2} \sum_{k \in I} \left[y_k - \left(x_1 + \frac{k}{v_k x_2 + w_k x_3} \right) \right]^2$$

$$x = (x_1, x_2, x_3), \quad y = (y_1, y_2, \ldots, y_{15})$$

where $v_k = 16 - k$, $w_k = \min(k, v_k)$ $(k \in I)$, and

$$y = (0.14, 0.18, 0.22, 0.25, 0.29, 0.32, 0.35, 0.39, 0.37, 0.58, 0.73, 0.96, 1.34, 2.10, 4.39)$$

The minimum function value $f(x^*) = 4.107\text{E}{-3}$ is at the point $(0.08, 1.13, 2.34)$. The starting point $x^0 = (1, 1, 1)$ is used. This problem is identical to the example "Using the DATA= Option" in Chapter 6, "The NLP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The following statements use the READ DATA statement to input parameter values and the CREATE DATA statement to save the solution in a SAS data set:

```
data bard;
   input y @@;
   datalines;
.14  .18  .22  .25  .29  .32  .35  .39
.37  .58  .73  .96 1.34 2.10 4.39
;
proc optmodel;
   set I = 1..15;
   number y{I};
   read data bard into [_n_] y;
   number v{k in I} = 16 - k;
   number w{k in I} = min(k, v[k]);
   var x{1..3} init 1;
   min f = 0.5*
      sum{k in I}
         (y[k] - (x[1] + k /
                  (v[k]*x[2] + w[k]*x[3])))**2;
   solve;
   print x;
   create data xdata from [i] xd=x;
```

In these statements the values for parameter y are read from the BARD data set. The set I indexes the terms of the objective in addition to the y array.

The preceding statements define two utility parameters that contain coefficients used in the objective function. These coefficients could have been defined in the expression for the objective, f, but it was convenient to give them names and simplify the objective expression.

The result is shown in Output 5.2.1.

Output 5.2.1 Bard Function Solution

[1]	x
1	0.08241
2	1.13303
3	2.34370

The final CREATE DATA statement saves the solution values determined by the solver into the data set XDATA. The data set contains an observation for each x index. Each observation contains two variables. The output variable i contains the index, while xd contains the value for the indexed entry in the array x. The resulting data can be seen by using the PRINT procedure as follows:

```
proc print data=xdata;
run;
```

The output from PROC PRINT is shown in Output 5.2.2.

Output 5.2.2 Output Data Set Contents

Obs	i	xd
1	1	0.08241
2	2	1.13303
3	3	2.34370

Example 5.3: Model Construction

This example uses PROC OPTMODEL features to simplify the construction of a mathematically formulated model. The model is based on the example "An Assignment Problem" in Chapter 4, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). A single invocation of PROC OPTMODEL replaces several steps in the PROC LP statements.

The model assigns production of various grades of cloth to a set of machines in order to maximize profit while meeting customer demand. Each machine has different capacities to produce the various grades of cloth. (See the PROC LP example "An Assignment Problem" for more details.) The mathematical formulation, where x_{ijk} represents the amount of cloth of grade j to produce on machine k for customer i, follows:

$$
\begin{array}{lll}
\max & \sum_{ijk} r_{ijk} x_{ijk} & \\
\text{subject to} & \sum_{k} x_{ijk} = d_{ij} & \text{for all } i \text{ and } j \\
 & \sum_{ij} c_{jk} x_{ijk} \leq a_k & \text{for all } k \\
 & x_{ijk} \geq 0 & \text{for all } i,\ j,\ \text{and } k
\end{array}
$$

The OBJECT, DEMAND, and RESOURCE data sets are the same as in the PROC LP example. A new data set, GRADE, is added to help separate the data from the model.

```
title 'An Assignment Problem';

data grade(drop=i);
   do i = 1 to 6;
      grade = 'grade'||put(i,1.);
      output;
   end;
run;

data object;
   input machine customer
         grade1 grade2 grade3 grade4 grade5 grade6;
   datalines;
1 1 102 140 105 105 125 148
1 2 115 133 118 118 143 166
1 3  70 108  83  83  88  86
1 4  79 117  87  87 107 105
1 5  77 115  90  90 105 148
2 1 123 150 125 124 154   .
2 2 130 157 132 131 166   .
2 3 103 130 115 114 129   .
2 4 101 128 108 107 137   .
2 5 118 145 130 129 154   .
3 1  83   .   .  97 122 147
3 2 119   .   . 133 163 180
3 3  67   .   .  91 101 101
3 4  85   .   . 104 129 129
3 5  90   .   . 114 134 179
4 1 108 121  79   . 112 132
4 2 121 132  92   . 130 150
```

```
4 3  78  91  59   .  77  72
4 4 100 113  76   . 109 104
4 5  96 109  77   . 105 145
;

data demand;
   input customer
         grade1 grade2 grade3 grade4 grade5 grade6;
   datalines;
1 100 100 150  150  175  250
2 300 125 300  275  310  325
3 400   0 400  500  340    0
4 250   0 750  750    0    0
5   0 600 300    0  210  360
;

data resource;
   input machine
         grade1 grade2 grade3 grade4 grade5 grade6 avail;
   datalines;
1 .250 .275 .300  .350  .310  .295  744
2 .300 .300 .305  .315  .320  .     244
3 .350 .    .     .320  .315  .300  790
4 .280 .275 .260  .     .250  .295  672
;
```

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called SOLUTION:

```
proc optmodel;
   /* declare index sets */
   set CUSTOMERS;
   set <str> GRADES;
   set MACHINES;

   /* declare parameters */
   num return {CUSTOMERS, GRADES, MACHINES} init 0;
   num demand {CUSTOMERS, GRADES};
   num cost {GRADES, MACHINES} init 0;
   num avail {MACHINES};

   /* read the set of grades */
   read data grade into GRADES=[grade];

   /* read the set of customers and their demands */
   read data demand
      into CUSTOMERS=[customer]
      {j in GRADES} <demand[customer,j]=col(j)>;

   /* read the set of machines, costs, and availability */
   read data resource nomiss
      into MACHINES=[machine]
      {j in GRADES} <cost[j,machine]=col(j)>
      avail;
```

```
   /* read objective data */
   read data object nomiss
      into [machine customer]
      {j in GRADES} <return[customer,j,machine]=col(j)>;

   /* declare the model */
   var AmountProduced {CUSTOMERS, GRADES, MACHINES} >= 0;
   max TotalReturn = sum {i in CUSTOMERS, j in GRADES, k in MACHINES}
      return[i,j,k] * AmountProduced[i,j,k];
   con req_demand {i in CUSTOMERS, j in GRADES}:
      sum {k in MACHINES} AmountProduced[i,j,k] = demand[i,j];
   con req_avail {k in MACHINES}:
      sum {i in CUSTOMERS, j in GRADES}
         cost[j,k] * AmountProduced[i,j,k] <= avail[k];

   /* call the solver and save the results */
   solve;
   create data solution
      from [customer grade machine] = {i in CUSTOMERS, j in GRADES,
         k in MACHINES: AmountProduced[i,j,k].sol ne 0}
      amount=AmountProduced;

   /* print optimal solution */
   print AmountProduced;
quit;
```

The statements use both numeric (NUM) and character (STR) index sets, which are populated from the corresponding data set variables in the READ DATA statements. The OPTMODEL parameters can be either single-dimensional (AVAIL) or multiple-dimensional (COST, DEMAND, RETURN). The RETURN and COST parameters are given initial values of 0, and the NOMISS option in the READ DATA statement tells PROC OPTMODEL to read only the nonmissing values from the input data sets. The model declaration is nearly identical to the mathematical formulation. The logical condition **`AmountProduced[i,j,k].sol ne 0`** in the CREATE DATA statement ensures that only the nonzero parts of the solution appear in the SOLUTION data set. In the PROC LP example, the creation of this data set required postprocessing of the PROC LP output data set.

The solver produces the following problem summary and solution summary:

Output 5.3.1 LP Solver Result

An Assignment Problem

Problem Summary	
Objective Sense	Maximization
Objective Function	TotalReturn
Objective Type	Linear
Number of Variables	120
Bounded Above	0
Bounded Below	120
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	34
Linear LE (<=)	4
Linear EQ (=)	30
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	220

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	TotalReturn
Solution Status	Optimal
Objective Value	871426.03763
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	59
Presolve Time	0.00
Solution Time	0.00

The SOLUTION data set can be processed by PROC TABULATE as follows to create a compact representation of the solution:

```
proc tabulate data=solution;
   class customer grade machine;
   var   amount;
   table (machine*customer), (grade*amount=''*sum='');
run;
```

These statements produce the table shown in Output 5.3.2.

Output 5.3.2 An Assignment Problem

An Assignment Problem

		grade					
		grade1	grade2	grade3	grade4	grade5	grade6
machine	customer						
1	1	.	100.00	150.00	150.00	175.00	250.00
	2	.	.	300.00	.	.	.
	3	.	.	256.72	210.31	.	.
	4	.	.	750.00	.	.	.
	5	.	92.27	.	.	.	.
2	3	.	.	143.28	.	340.00	.
	5	.	.	300.00	.	.	.
3	2	.	.	.	275.00	310.00	325.00
	3	.	.	.	289.69	.	.
	4	.	.	.	750.00	.	.
	5	.	.	.	.	210.00	360.00
4	1	100.00	.	.	.	.	.
	2	300.00	125.00	.	.	.	.
	3	400.00	.	.	.	.	.
	4	250.00	.	.	.	.	.
	5	.	507.73	.	.	.	.

Example 5.4: Set Manipulation

This example demonstrates PROC OPTMODEL set manipulation operators. These operators are used to compute the set of primes up to a given limit. This example does not solve an optimization problem, but similar set manipulation could be used to set up an optimization model. Here are the statements:

```
proc optmodel;
   number maxprime;  /* largest number to consider */
   set composites =
       union {i in 3..sqrt(maxprime) by 2} i*i..maxprime by 2*i;
   set primes = {2} union (3..maxprime by 2 diff composites);
   maxprime = 500;
   put primes;
```

The set composites contains the odd composite numbers up to the value of the parameter maxprime. The even numbers are excluded here to reduce execution time and memory requirements. The UNION aggregation operation is used in the definition to combine the sets of odd multiples of i for $i = 3, 5, \ldots$. Any composite number less than the value of the parameter maxprime has a divisor $\leq \sqrt{\text{maxprime}}$, so the range of i can be limited. The set of multiples of i can also be started at $i \times i$ since smaller multiples are found in the set of multiples for a smaller index.

You can then define the set primes. The odd primes are determined by using the DIFF operator to remove the composites from the set of odd numbers no greater than the parameter maxprime. The UNION operator adds the single even prime, 2, to the resulting set of primes.

The PUT statement produces the result in Output 5.4.1.

Output 5.4.1 Primes less than or equal to 500

```
{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,
107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,211,
223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,331,
337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,433,439,443,449,
457,461,463,467,479,487,491,499}
```

Note that you were able to delay the definition of the value of the parameter maxprime until just before the PUT statement. Since the defining expressions of the SET declarations are handled symbolically, the value of maxprime is not necessary until you need the value of the set primes. Because the sets composites and primes are defined symbolically, their values reflect any changes to the parameter maxprime. You can see this update by appending the following statements to the preceding statements:

```
maxprime = 50;
put primes;
```

The additional statements produce the results in Output 5.4.2. The value of the set primes has been recomputed to reflect the change to the parameter maxprime.

Output 5.4.2 Primes less than or equal to 50

```
{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
```

Example 5.5: Multiple Subproblems

Many important optimization problems cannot be solved directly using a standard solver, either because the problem has constraints that cannot be modeled directly or because the resulting model would be too large to be practical. For these types of problems, you can use PROC OPTMODEL to synthesize solution methods by using a combination of the existing solvers and the modeling language programming constructions. This example demonstrates the use of multiple subproblems to solve the cutting stock problem.

The cutting stock problem determines how to efficiently cut raw stock into finished widths based on the demands for the final product. Consider the example from page 195 of Chvátal (1983), where raw stock has a width of 100 inches and the demands are shown in Table 5.18.

Table 5.18 Cutting Stock Demand

Finished Width	Demand
45 inches	97
35 inches	610
31 inches	395
14 inches	211

A portion of the demand can be satisfied using a cutting pattern. For example, with the demands in Table 5.18 a possible pattern cuts one final of width 35 inches, one final of width 31 inches, and two finals of width 14 inches. This gives:

$$100 = 0 * 45 + 1 * 35 + 1 * 31 + 2 * 14 + \text{waste}.$$

The cutting stock problem can be formulated as follows, where x_j represents the number of times pattern j appears, a_{ij} represents the number of times demand item i appears in pattern j, d_i is the demand for item i, w_i is the width of item i, N represents the set of patterns, M represents the set of items, and W is the width of the raw stock:

$$\begin{array}{ll} \text{minimize} & \sum_{j \in N} x_j \\ \text{subject to} & \sum_{j \in N} a_{ij} x_j \geq d_i \quad \text{for all } i \in M \\ & x_j \text{ integer}, \geq 0 \quad \text{for all } j \in N \end{array}$$

Also for each feasible pattern j you must have:

$$\sum_{i \in M} w_i a_{ij} \leq W$$

The difficulty with this formulation is that the number of patterns can be very large, with too many columns x_j to solve efficiently. But you can use column generation, as described on page 198 of Chvátal (1983), to generate a smaller set of useful patterns, starting from an initial feasible set.

The dual variables, π_i, of the demand constraints are used to price out the columns. From linear programming (LP) duality theory, a column that improves the primal solution must have a negative reduced cost. For this problem the reduced cost for column x_j is

$$1 - \sum_{i \in M} \pi_i a_{ij}$$

Using this observation produces a knapsack subproblem:

$$\begin{array}{ll} \text{minimize} & 1 - \sum_{i \in M} \pi_i a_i \\ \text{subject to} & \sum_{i \in M} w_i a_i \leq W \\ & a_i \text{ integer}, \geq 0 \quad \text{for all } j \in N \end{array}$$

where the objective is equivalent to:

$$\text{maximize} \quad \sum_{i \in M} \pi_i a_i$$

The pattern is useful if the associated reduced cost is negative:

$$1 - \sum_{i \in M} \pi_i a_i^* < 0$$

So you can use the following steps to generate the patterns and solve the cutting stock problem:

1. Initialize a set of trivial (one item) patterns.
2. Solve the problem using the LP solver.
3. Minimize the reduced cost using a knapsack solver.

4. Include the new pattern if the reduced cost is negative.

5. Repeat steps 2 through 4 until there are no more negative reduced cost patterns.

These steps are implemented in the following statements. Since adding columns preserves primal feasibility, the statements use the primal simplex solver to take advantage of a warm start. The statements also solve the LP relaxation of the problem, but you want the integer solution. So the statements finish by using the MILP solver with the integer restriction applied. The result is not guaranteed to be optimal, but lower and upper bounds can be provided for the optimal objective.

```
/* cutting-stock problem */
/* uses delayed column generation from
   Chvatal's Linear Programming (1983), page 198 */

%macro csp(capacity);
proc optmodel printlevel=0;
   /* declare parameters and sets */
   num capacity = &capacity;
   set ITEMS;
   num demand {ITEMS};
   num width  {ITEMS};
   num num_patterns init card(ITEMS);
   set PATTERNS = 1..num_patterns;
   num a {ITEMS, PATTERNS};
   num c {ITEMS} init 0;
   num epsilon = 1E-6;

   /* read input data */
   read data indata into ITEMS=[_N_] demand width;

   /* generate trivial initial columns */
   for {i in ITEMS, j in PATTERNS}
      a[i,j] = (if (i = j) then floor(capacity/width[i]) else 0);

   /* define master problem */
   var x {PATTERNS} >= 0 integer;
   minimize NumberOfRaws = sum {j in PATTERNS} x[j];
   con demand_con {i in ITEMS}:
      sum {j in PATTERNS} a[i,j] * x[j] >= demand[i];
   problem Master include x NumberOfRaws demand_con;

   /* define column generation subproblem */
   var y {ITEMS} >= 0 integer;
   maximize KnapsackObjective = sum {i in ITEMS} c[i] * y[i];
   con knapsack_con:
      sum {i in ITEMS} width[i] * y[i] <= capacity;
   problem Knapsack include y KnapsackObjective knapsack_con;

   /* main loop */
   do while (1);
      print _page_ a;

      /* master problem */
```

```
      /* minimize sum_j x[j]
         subj. to sum_j a[i,j] * x[j] >= demand[i]
                  x[j] >= 0 and integer */
      use problem Master;
      put "solve master problem";
      solve with lp relaxint /
         presolver=none solver=ps basis=warmstart printfreq=1;
      print x;
      print demand_con.dual;
      for {i in ITEMS} c[i] = demand_con[i].dual;

      /* knapsack problem */
      /* maximize sum_i c[i] * y[i]
         subj. to sum_i width[i] * y[i] <= capacity
                  y[i] >= 0 and integer */
      use problem Knapsack;
      put "solve column generation subproblem";
      solve with milp / printfreq=0;
      for {i in ITEMS} y[i] = round(y[i]);
      print y;
      print KnapsackObjective;

      if KnapsackObjective <= 1 + epsilon then leave;

      /* include new pattern */
      num_patterns = num_patterns + 1;
      for {i in ITEMS} a[i,num_patterns] = y[i];
   end;

   /* solve IP, using rounded-up LP solution as warm start */
   use problem Master;
   for {j in PATTERNS} x[j] = ceil(x[j].sol);
   put "solve (restricted) master problem as IP";
   solve with milp / primalin;

   /* cleanup solution and save to output data set */
   for {j in PATTERNS} x[j] = round(x[j].sol);
   create data solution from [pattern]={j in PATTERNS: x[j] > 0}
      raws=x {i in ITEMS} <col('i'||i)=a[i,j]>;
quit;
%mend csp;

/* Chvatal, p.199 */
data indata;
   input demand width;
   datalines;
78 25.5
40 22.5
30 20
30 15
;
run;
%csp(91)
/* LP solution is integer */
```

```
/* Chvatal, p.195 */
data indata;
   input demand width;
   datalines;
 97 45
610 36
395 31
211 14
;
run;
%csp(100)
/* LP solution is fractional */
```

The contents of the output data set for the second problem instance are shown in Output 5.5.1.

Output 5.5.1 Cutting Stock Solution

Obs	pattern	raws	i1	i2	i3	i4
1	1	49	2	0	0	0
2	2	100	0	2	0	0
3	5	106	0	2	0	2
4	6	198	0	1	2	0

Example 5.6: Traveling Salesman Problem

This example demonstrates the use of the SUBMIT statement to execute a block of SAS statements from within a PROC OPTMODEL session. In this case, the SUBMIT block calls the GPLOT procedure to display intermediate results during the solution of an instance of the traveling salesman problem (TSP). The problem is described in Example 8.4. For an example of how to use PROC OPTNET to solve the TSP, see "Traveling Salesman Problem Applied to a Simple Directed Graph" (Chapter 2, *SAS/OR User's Guide: Network Optimization Algorithms*).

The following DATA step converts a TSPLIB instance of type EUC_2D into a SAS data set that contains the coordinates of the vertices:

```
/* convert the TSPLIB instance into a data set */
data tspData(drop=H);
   infile "&tsplib";
   input H $1. @;
   if H not in ('N','T','C','D','E');
   input @1 var1-var3;
run;
```

The following macro generates plots of the solution and objective value:

```
%macro plotTSP;
   /* create Annotate data set to draw subtours */
   data anno(drop=xi yi xj yj);
      %SYSTEM(2, 2, 2);
      set solData(keep=xi yi xj yj);
      %LINE(xi, yi, xj, yj, *, 1, 1);
   run;

   title1 h=2 "TSP: Iter = &i, Objective = &&obj&i";
   title2;

   proc gplot data=tspData anno=anno;
      axis1 label=none;
      symbol1 value=dot interpol=none
      pointlabel=("#var1" nodropcollisions height=1) cv=black;
      plot var3*var2 / haxis=axis1 vaxis=axis1;
   run;
   quit;
%mend plotTSP;

%annomac;
```

The following PROC OPTMODEL statements solve the TSP by using the subtour formulation and iteratively adding subtour constraints. The SUBMIT statement calls the **%plotTSP** macro to plot the solution and objective value at each stage.

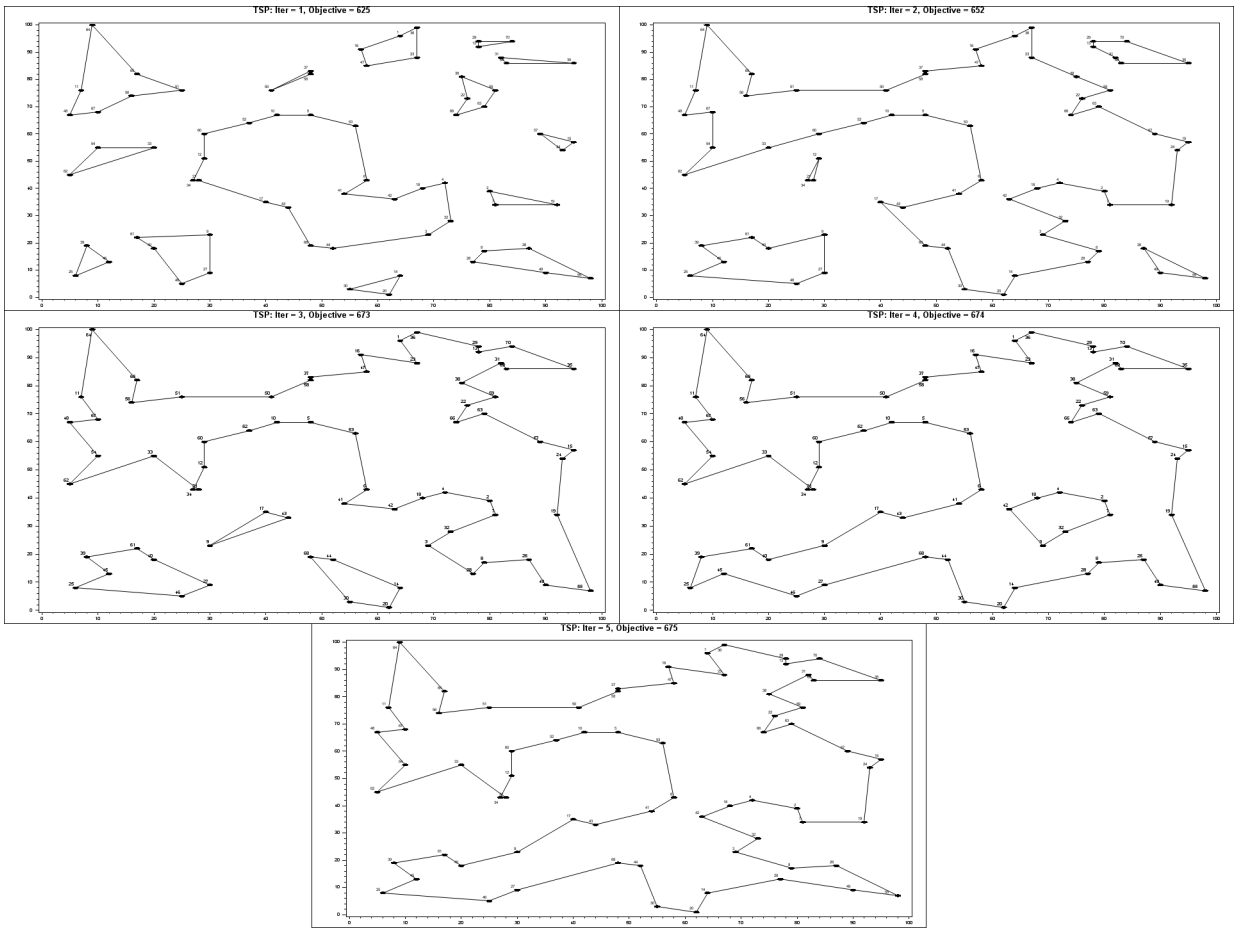
```
/* iterative solution using the subtour formulation */
proc optmodel;
   set VERTICES;
   set EDGES = {i in VERTICES, j in VERTICES: i > j};
   num xc {VERTICES};
   num yc {VERTICES};

   num numsubtour init 0;
   set SUBTOUR {1..numsubtour};

   /* read in the instance and customer coordinates (xc, yc) */
   read data tspData into VERTICES=[var1] xc=var2 yc=var3;

   /* the cost is the euclidean distance rounded to the nearest integer */
   num c {<i,j> in EDGES}
      init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);

   var x {EDGES} binary;

   /* minimize the total cost */
   min obj =
       sum {<i,j> in EDGES} c[i,j] * x[i,j];

   /* each vertex has exactly one in-edge and one out-edge */
   con two_match {i in VERTICES}:
      sum {j in VERTICES: i > j} x[i,j]
      + sum {j in VERTICES: i < j} x[j,i] = 2;
```

```
/* no subtours (these constraints are generated dynamically) */
con subtour_elim {s in 1..numsubtour}:
   sum {<i,j> in EDGES: (i in SUBTOUR[s] and j not in SUBTOUR[s])
      or (i not in SUBTOUR[s] and j in SUBTOUR[s])} x[i,j] >= 2;

/* this starts the algorithm to find violated subtours */
set <num,num> EDGES1;
set INITVERTICES = setof{<i,j> in EDGES1} i;
set VERTICES1;
set NEIGHBORS;
set <num,num> CLOSURE;
num component {INITVERTICES};
num numcomp  init 2;
num iter     init 1;
call symput('i',trim(left(put(round(iter),best.))));
num numiters init 1;

/* initial solve with just matching constraints */
solve;
call symput(compress('obj'||put(iter,best.)),
           trim(left(put(round(obj),best.))));


/* create a data set for use by PROC GPLOT */
create data solData from
   [i j]={<i,j> in EDGES: x[i,j].sol > 0.5}
   xi=xc[i] yi=yc[i] xj=xc[j] yj=yc[j];
submit;
   %plotTSP;
endsubmit;
/* while the solution is disconnected, continue */
do while (numcomp > 1);
   iter = iter + 1;
   call symput('i',trim(left(put(round(iter),best.))));

   /* find connected components of support graph   */
   EDGES1 = {<i,j> in EDGES: round(x[i,j].sol) = 1};
   EDGES1 = EDGES1 union {setof {<i,j> in EDGES1} <j,i>};
   VERTICES1 = INITVERTICES;
   CLOSURE = EDGES1;

   for {i in INITVERTICES} component[i] = 0;
   for {i in VERTICES1} do;
      NEIGHBORS = slice(<i,*>,CLOSURE);
      CLOSURE = CLOSURE union (NEIGHBORS cross NEIGHBORS);
   end;

   numcomp = 0;
   do while (card(VERTICES1) > 0);
      numcomp = numcomp + 1;
      for {i in VERTICES1} do;
         NEIGHBORS = slice(<i,*>,CLOSURE);
         for {j in NEIGHBORS} component[j] = numcomp;
```

```
            VERTICES1 = VERTICES1 diff NEIGHBORS;
            leave;
         end;
      end;

      if numcomp = 1 then leave;

      numiters = iter;
      numsubtour = numsubtour + numcomp;
      for {comp in 1..numcomp} do;
         SUBTOUR[numsubtour-numcomp+comp]
           = {i in VERTICES: component[i] = comp};
      end;

      solve;
      call symput(compress('obj'||put(iter,best.)),
                  trim(left(put(round(obj),best.))));

      /* create a data set for use by PROC GPLOT */
      create data solData from
         [i j]={<i,j> in EDGES: x[i,j].sol > 0.5}
         xi=xc[i] yi=yc[i] xj=xc[j] yj=yc[j];

      call symput('numiters',put(numiters,best.));
      submit;
         %plotTSP;
      endsubmit;
   end;
quit;
```

The plot in Output 5.6.1 shows the solution and objective value at each stage. Each stage restricts some subset of subtours. When you reach the final stage, you have a valid tour.

Output 5.6.1 Iterative Solution of Traveling Salesman Problem



Example 5.7: Sparse Modeling

This example demonstrates how to rewrite certain models for more efficient processing. Sometimes optimization models that run out of memory during problem generation can be rewritten to take advantage of sparsity to use memory more efficiently. This often occurs when a large array is modeled in a dense format but most of its entries are zeros. Usually, the array provides problem coefficients or it contains optimization variables.

The model for this example solves the facility location problem that is described in Example 8.3. This example is concerned with the resources that are required for PROC OPTMODEL problem generation and solver initialization. So the size of the problem has been increased, but the model has also been modified to make it easier to solve. In order to handle the larger problem size, the model eliminates a large number of the potential assignments of customers to facilities based on distance, making the problem sparse.

The following code generates a random instance of the facility location problem:

```
%let NumCustomers  = 1500;
%let NumSites      = 250;
%let SiteCapacity  = 50;
%let MaxDemand     = 10;
%let xmax          = 200;
%let ymax          = 100;
%let seed          = 938;

/* generate random customer locations */
data cdata(drop=i);
   length name $8;
   call streaminit(&seed);
   do i = 1 to &NumCustomers;
      name = compress('C'||put(i,best.));
      x = rand('UNIFORM') * &xmax;
      y = rand('UNIFORM') * &ymax;
      demand = 1;
      output;
   end;
run;

/* generate random site locations and fixed charge */
data sdata(drop=i);
   length name $8;
   call streaminit(&seed);
   do i = 1 to &NumSites;
      name = compress('SITE'||put(i,best.));
      x = rand('UNIFORM') * &xmax;
      y = rand('UNIFORM') * &ymax;
      fixed_charge = 300 * (abs(&xmax/2-x)/&xmax + abs(&ymax/2-y)/&ymax);
      output;
   end;
run;
```

The following code uses a dense version of the facility location model. This model is equivalent to the model from Example 8.3 except for the added constraint distance_at_most_30. This constraint eliminates from consideration the assignment of customers to facilities over long distances by forcing the corresponding Assign variables to 0.

```
proc optmodel;
   performance details;
   profile on percent=0.1;
   set <str> CUSTOMERS;
   set <str> SITES init {};

   /* x and y coordinates of CUSTOMERS and SITES */
   num x {CUSTOMERS union SITES};
   num y {CUSTOMERS union SITES};
   num demand {CUSTOMERS};
   num fixed_charge {SITES};

   /* distance from customer i to site j */
   num dist {i in CUSTOMERS, j in SITES}
```

```
      = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);

   read data cdata into CUSTOMERS=[name] x y demand;
   read data sdata into SITES=[name] x y fixed_charge;

   var Assign {CUSTOMERS, SITES} binary;
   var Build {SITES} binary;

   min CostNoFixedCharge
       = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j];
   min CostFixedCharge
       = CostNoFixedCharge + sum {j in SITES} fixed_charge[j] * Build[j];

   /* each customer assigned to exactly one site */
   con assign_def {i in CUSTOMERS}:
      sum {j in SITES} Assign[i,j] = 1;

   /* if customer i assigned to site j, then facility must be built at j */
   con link {i in CUSTOMERS, j in SITES}:
      Assign[i,j] <= Build[j];

   /* each site can handle at most &SiteCapacity demand */
   con capacity {j in SITES}:
      sum {i in CUSTOMERS} demand[i] * Assign[i,j] <=
         &SiteCapacity * Build[j];

   /* do not assign customer to site more than 30 units away */
   con distance_at_most_30 {i in CUSTOMERS, j in SITES: dist[i,j] > 30}:
      Assign[i,j] = 0;

   /* solve the MILP */
   solve with milp/timetype=real;

quit;
```

If you inspect the log after running the preceding code, then you will see that the MILP presolver has pruned down the problem size considerably. If you also run the code with the SAS option FULLSTIMER enabled on a 64-bit system, then you will notice that about 1.3GB of memory is required for the PROC OPTMODEL step when you are running on a single CPU.

The solution and timing results for the dense model are shown in Output 5.7.1. The PERFORMANCE DETAILS statement from the model requests display of the task timing table for the SOLVE statement. The PROFILE ON statement requests further timing details, including evaluation time for declarations used by problem generation.

Output 5.7.1 Dense Model Results

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	CostFixedCharge
Solution Status	Optimal within Relative Gap
Objective Value	18001.14035
Relative Gap	0.000098081
Absolute Gap	1.7653963848
Primal Infeasibility	1.14871E-12
Bound Infeasibility	9.9999999E-7
Integer Infeasibility	1E-6
Best Bound	17999.374954
Nodes	5
Iterations	40131
Presolve Time	1.20
Solution Time	62.73

Procedure Task Timing		
Task	**Time (sec.)**	**Time**
Problem Generation	4.57	6.64%
Solver Initialization	0.90	1.30%
Code Generation	0.06	0.08%
Presolve	1.20	1.75%
Root Node Processing	50.70	73.62%
Branch And Cut	9.32	13.54%
Synchronization	0.20	0.30%
Idle	1.13	1.64%
Other Tasks	0.17	0.25%
Solver Postprocessing	0.62	0.89%

Profile Information						
Item	**Line**	**Col.**	**Execution Count**	**Net Time (sec.)**	**Wait Time (sec.)**	**% Total Time**
SOLVE	3546	4	1	64.85	2.17	85.9%
Constraint distance_at_most_30	3542	8		5.59	0.00	7.4%
Constraint link	3533	8		3.22	0.00	4.3%
Number dist	3514	8		0.52	0.00	0.7%
Constraint capacity	3537	8		0.38	0.00	0.5%
Min CostNoFixedCharge	3523	8		0.37	0.00	0.5%
Var Assign	3520	8		0.29	0.00	0.4%
Constraint assign_def	3529	8		0.28	0.00	0.4%
Other profiled items				0.01	0.00	0.0%

Note: Total profiled time is 75.51 seconds.

The best approach for reducing the memory requirements is to eliminate the Assign variables that are always going to be 0. This is accomplished in the following sparse version of the code. Instead of indexing Assign over the crossproduct of CUSTOMERS and SITES, now the code defines a new set of pairs that satisfy the distance requirement, CUSTOMERS_SITES. This set replaces the constraint distance_at_most_30 in the dense model. The objective and constraints have been modified to use the new indexing scheme, with implicit set slicing (as described in the section "More on Index Sets" on page 159) for constraints assign_def and capacity.

```
proc optmodel;
   performance details;
   profile on percent=0.1;
   set <str> CUSTOMERS;
   set <str> SITES init {};

   /* x and y coordinates of CUSTOMERS and SITES */
   num x {CUSTOMERS union SITES};
   num y {CUSTOMERS union SITES};
   num demand {CUSTOMERS};
   num fixed_charge {SITES};

   /* distance from customer i to site j */
   num dist {i in CUSTOMERS, j in SITES}
       = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);

   read data cdata into CUSTOMERS=[name] x y demand;
   read data sdata into SITES=[name] x y fixed_charge;

   set CUSTOMERS_SITES = {i in CUSTOMERS, j in SITES: dist[i,j] <= 30};
   var Assign {CUSTOMERS_SITES} binary;
   var Build {SITES} binary;

   min CostNoFixedCharge
       = sum {<i,j> in CUSTOMERS_SITES} dist[i,j] * Assign[i,j];
   min CostFixedCharge
       = CostNoFixedCharge + sum {j in SITES} fixed_charge[j] * Build[j];

   /* each customer assigned to exactly one site */
   con assign_def {i in CUSTOMERS}:
      sum {<(i),j> in CUSTOMERS_SITES} Assign[i,j] = 1;

   /* if customer i assigned to site j, then facility must be built at j */
   con link {<i,j> in CUSTOMERS_SITES}:
      Assign[i,j] <= Build[j];

   /* each site can handle at most &SiteCapacity demand */
   con capacity {j in SITES}:
      sum {<i,(j)> in CUSTOMERS_SITES} demand[i] * Assign[i,j] <=
         &SiteCapacity * Build[j];

   /* solve the MILP */
   solve with milp/timetype=real;

quit;
```

The log from running the preceding code shows that the MILP presolver does not find anything to improve with this version of the model. On a 64-bit system, the FULLSTIMER option shows that memory requirements have been reduced to about 580MB when you are running on a single CPU, less than half the requirements of the previous model.

The solution and timing results for the dense model are shown in Output 5.7.2. Note that the dense model (Output 5.7.1) and the sparse model (Output 5.7.2) are equivalent after presolver processing and generate the same result using similar amounts of solver time. On the other hand, problem generation time is significantly reduced as are other times including presolve time. Both models used the solver option TIMETYPE=REAL so that all times are reported in seconds of real time. You can see from the "Profile Information" table that the overhead associated with problem declarations has been significantly reduced.

Output 5.7.2 Sparse Model Results

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	CostFixedCharge
Solution Status	Optimal within Relative Gap
Objective Value	18001.14035
Relative Gap	0.000098081
Absolute Gap	1.7653963848
Primal Infeasibility	1.14871E-12
Bound Infeasibility	9.9999999E-7
Integer Infeasibility	1E-6
Best Bound	17999.374954
Nodes	5
Iterations	40131
Presolve Time	0.29
Solution Time	60.60

Procedure Task Timing		
Task	**Time (sec.)**	**Time**
Problem Generation	1.19	1.92%
Solver Initialization	0.10	0.17%
Code Generation	0.00	0.01%
Presolve	0.29	0.47%
Root Node Processing	49.82	80.44%
Branch And Cut	9.02	14.57%
Synchronization	0.20	0.32%
Idle	1.10	1.77%
Other Tasks	0.17	0.28%
Solver Postprocessing	0.04	0.06%

Output 5.7.2 *continued*

Profile Information

Item	Line	Col.	Execution Count	Net Time (sec.)	Wait Time (sec.)	% Total Time
SOLVE	3596	4	1	60.79	0.12	97.5%
Number dist	3567	8		0.49	0.00	0.8%
Set CUSTOMERS_SITES	3573	8		0.46	0.00	0.7%
Constraint link	3587	8		0.35	0.00	0.6%
Constraint assign_def	3583	8		0.08	0.00	0.1%
Constraint capacity	3591	8		0.07	0.00	0.1%
Other profiled items				0.08	0.00	0.1%

Note: Total profiled time is 62.32 seconds.

Example 5.8: Chemical Equilibrium

This example illustrates how to convert PROC NLP code that handles arrays into PROC OPTMODEL form. The following PROC NLP model finds an equilibrium state for a mixture of chemicals. The same model is used in "Example 7.8: Chemical Equilibrium" in Chapter 6, "The NLP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

```
proc nlp tech=tr pall;
   array c[10] -6.089 -17.164 -34.054  -5.914 -24.721
               -14.986 -24.100 -10.708 -26.662 -22.179;
   array x[10] x1-x10;
   min y;
   parms x1-x10 = .1;
   bounds 1.e-6 <= x1-x10;
   lincon 2. = x1 + 2. * x2 + 2. * x3 + x6 + x10,
          1. = x4 + 2. * x5 + x6 + x7,
          1. = x3 + x7  + x8 + 2. * x9 + x10;
   s = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10;
   y = 0.;
   do j = 1 to 10;
      y = y + x[j] * (c[j] + log(x[j] / s));
   end;
run;
```

The following statements show a corresponding PROC OPTMODEL model:

```
proc optmodel;
   set CMP = 1..10;
   number c{CMP} = [-6.089 -17.164 -34.054  -5.914 -24.721
                    -14.986 -24.100 -10.708 -26.662 -22.179];
   var x{CMP} init 0.1 >= 1.e-6;
   con 2. = x[1] + 2. * x[2] + 2. * x[3] + x[6] + x[10],
       1. = x[4] + 2. * x[5] + x[6] + x[7],
       1. = x[3] + x[7]  + x[8] + 2. * x[9] + x[10];
   /* replace the variable s in the PROC NLP model */
   impvar s = sum{i in CMP} x[i];
   min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
   solve;
   print x y;
```

The PROC OPTMODEL model uses the set CMP to represent the set of compounds, which are numbered 1 to 10 in the example. The array c was initialized by using the equivalent PROC OPTMODEL syntax. The individual array locations could also have been initialized by assignment or by READ DATA statements.

The VAR declaration for variable x combines the VAR and BOUNDS statements of the PROC NLP model. The index set of the array is based on the set of compounds CMP to simplify changes to the model.

The linear constraints are similar in form to the PROC NLP model. However, the PROC OPTMODEL version uses the array form of the variable names because it treats arrays as distinct variables, not as aliases of lists of scalar variables.

The implicit variable s replaces the intermediate variable of the same name in the PROC NLP model. This is an example of translating an intermediate variable from the other models to PROC OPTMODEL. An alternative way is to use an additional constraint for every intermediate variable. Instead of declaring objective s as in the preceding statements, you can use the following statements:

```
. . .
var s;
con s = sum{i in CMP} x[i];
. . .
```

Note that this alternative formulation passes an extra variable and constraint to the solver. This formulation can sometimes be solved more efficiently, depending on the characteristics of the model.

The PROC OPTMODEL version uses a SUM operator over the set CMP, which enhances the flexibility of the model to accommodate possible changes in the set of compounds.

In the PROC NLP model, the objective function y is determined by an explicit loop. The DO loop in PROC NLP is replaced by a SUM aggregation operation in PROC OPTMODEL. The accumulation in the PROC NLP model is now performed by PROC OPTMODEL by using the SUM operator.

This PROC OPTMODEL model can be generalized further. Note that the array initialization and constraints assume a fixed set of compounds. You can rewrite the model to handle an arbitrary number of compounds and chemical elements. The new model loads the linear constraint coefficients from a data set along with the objective coefficients for the parameter c, as follows:

```
data comp;
   input c a_1 a_2 a_3;
   datalines;
-6.089   1 0 0
-17.164  2 0 0
-34.054  2 0 1
-5.914   0 1 0
-24.721  0 2 0
-14.986  1 1 0
-24.100  0 1 1
-10.708  0 0 1
-26.662  0 0 2
-22.179  1 0 1
;

data atom;
   input b @@;
   datalines;
```

```
2. 1. 1.
;

proc optmodel;
   set CMP;
   set ELT;
   number c{CMP};
   number a{ELT,CMP};
   number b{ELT};
   read data atom into ELT=[_n_] b;
   read data comp into CMP=[_n_]
        c {i in ELT} < a[i,_n_]=col("a_"||i) >;
   var x{CMP} init 0.1 >= 1.e-6;
   con bal{i in ELT}: b[i] = sum{j in CMP} a[i,j]*x[j];
   impvar s = sum{i in CMP} x[i];
   min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
   print a b;
   solve;
   print x;
```

This version adds coefficients for the linear constraints to the COMP data set. The data set variable a_*n* represents the number of atoms in the compound for element *n*. The READ DATA statement for COMP uses the iterated column syntax to read each of the data set variables a_*n* into the appropriate location in the array a. In this example the expanded data set variable names are a_1, a_2, and a_3.

The preceding version also adds a new set, ELT, of chemical elements and a numeric parameter, b, that represents the left-hand side of the linear constraints. The data values for the parameters ELT and b are read from the data set ATOM. The model can handle varying sets of chemical elements because of this extra data set and the new parameters.

The linear constraints have been converted to a single, indexed family of constraints. One constraint is applied for each chemical element in the set ELT. The constraint expression uses a simple form that applies generally to linear constraints. The following PRINT statement in the model shows the values that are read from the data sets to define the linear constraints:

```
print a b;
```

The PRINT statements in the model produce the results shown in Output 5.8.1.

Output 5.8.1 PROC OPTMODEL Output

	a									
	1	2	3	4	5	6	7	8	9	10
1	1	2	2	0	0	1	0	0	0	1
2	0	0	0	1	2	1	1	0	0	0
3	0	0	1	0	0	0	1	1	2	1

[1]	b
1	2
2	1
3	1

Output 5.8.1 *continued*

[1]	x
1	0.04066848
2	0.14773067
3	0.78315260
4	0.00141459
5	0.48524616
6	0.00069358
7	0.02739955
8	0.01794757
9	0.03731444
10	0.09687143

In the preceding model, the chemical elements and compounds are designated by numbers. So in the PRINT output, for example, the row that is labeled "3" represents the amount of the compound H_2O. PROC OPTMODEL is capable of using more symbolic strings to designate array indices. The following version of the model uses strings to index arrays:

```
data comp;
   input name $ c a_h a_n a_o;
   datalines;
H     -6.089   1 0 0
H2    -17.164  2 0 0
H2O   -34.054  2 0 1
N     -5.914   0 1 0
N2    -24.721  0 2 0
NH    -14.986  1 1 0
NO    -24.100  0 1 1
O     -10.708  0 0 1
O2    -26.662  0 0 2
OH    -22.179  1 0 1
;
data atom;
   input name $ b;
   datalines;
H  2.
N  1.
O  1.
;
proc optmodel;
  set<string> CMP;
  set<string> ELT;
  number c{CMP};
  number a{ELT,CMP};
  number b{ELT};
  read data atom into ELT=[name] b;
  read data comp into CMP=[name]
       c {i in ELT} < a[i,name]=col("a_"||i) >;
  var x{CMP} init 0.1 >= 1.e-6;
  con bal{i in ELT}: b[i] = sum{j in CMP} a[i,j]*x[j];
  impvar s = sum{i in CMP} x[i];
  min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
  solve;
  print x;
```

In this model, the sets CMP and ELT are now sets of strings. The data sets provide the names of the compounds and elements. The names of the data set variables for atom counts in the data set COMP now include the chemical element symbol as part of their spelling. For example, the atom count for element H (hydrogen) is named a_h. Note that these changes did not require any modification to the specifications of the linear constraints or of the objective.

The PRINT statement in the preceding statements produces the results shown in Output 5.8.2. The indices of variable x are now strings that represent the actual compounds.

Output 5.8.2 PROC OPTMODEL Output with Strings

[1]	x
H	0.04066848
H2	0.14773067
H2O	0.78315260
N	0.00141459
N2	0.48524616
NH	0.00069358
NO	0.02739955
O	0.01794757
O2	0.03731444
OH	0.09687143

References

Abramowitz, M., and Stegun, I. A., eds. (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* 10th printing. New York: Dover.

Bazaraa, M. S., Sherali, H. D., and Shetty, C. M. (1993). *Nonlinear Programming: Theory and Algorithms*. New York: John Wiley & Sons.

Chvátal, V. (1983). *Linear Programming*. New York: W. H. Freeman.

Dennis, J. E., and Schnabel, R. B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall.

Fletcher, R. (1987). *Practical Methods of Optimization*. 2nd ed. Chichester, UK: John Wiley & Sons.

Nocedal, J., and Wright, S. J. (1999). *Numerical Optimization*. New York: Springer-Verlag.

Chapter 6
The Constraint Programming Solver

Contents

Overview: CLP Solver **194**
Getting Started: CLP Solver **194**
Send More Money 195
Eight Queens 197
Syntax: CLP Solver **199**
Functional Summary 199
SOLVE WITH CLP Statement 199
General Options 200
Predicates 202
Common Syntax Components 202
ALLDIFF Predicate 203
ELEMENT Predicate 204
GCC Predicate 205
LEXICO Predicate 206
PACK Predicate 207
REIFY Predicate 208
Details: CLP Solver **209**
Types of CSPs 209
Techniques for Solving CSPs 209
Differences between PROC OPTMODEL and PROC CLP 211
Macro Variable _OROPTMODEL_ 211
Examples: CLP Solver **214**
Example 6.1: Logic-Based Puzzles 214
Example 6.2: Alphabet Blocks Problem 221
Example 6.3: Work-Shift Scheduling Problem 224
Example 6.4: A Nonlinear Optimization Problem 228
Example 6.5: Car Painting Problem 231
Example 6.6: Scene Allocation Problem 233
Example 6.7: Car Sequencing Problem 237
Example 6.8: Balanced Incomplete Block Design 242
Example 6.9: Progressive Party Problem 246
References **253**

Overview: CLP Solver

You can use the constraint logic programming (CLP) solver in the OPTMODEL procedure to address finite-domain constraint satisfaction problems (CSPs) that have linear, logical, and global constraints. In addition to providing an expressive syntax for representing CSPs, the CLP solver features powerful built-in consistency routines and constraint propagation algorithms, a choice of nondeterministic search strategies, and controls for guiding the search mechanism. These features enable you to solve a diverse array of combinatorial problems.

Many important problems in areas such as artificial intelligence (AI) and operations research (OR) can be formulated as constraint satisfaction problems. A CSP is defined by a finite set of variables that take values from finite domains and by a finite set of constraints that restrict the values that the variables can simultaneously take.

A solution to a CSP is an assignment of values to the variables in order to satisfy all the constraints. The problem amounts to finding one or more solutions, or possibly determining that a solution does not exist.

A constraint satisfaction problem (CSP) can be defined as a triplet $\langle X, D, C \rangle$:

- $X = \{x_1, \ldots, x_n\}$ is a finite set of *variables*.
- $D = \{D_1, \ldots, D_n\}$ is a finite set of *domains*, where D_i is a finite set of possible values that the variable x_i can take. D_i is known as the *domain* of variable x_i.
- $C = \{c_1, \ldots, c_m\}$ is a finite set of *constraints* that restrict the values that the variables can simultaneously take.

The domains do not need to represent consecutive integers. For example, the domain of a variable could be the set of all even numbers in the interval [0, 100]. In PROC OPTMODEL, variables are always numeric. Therefore, if your problem contains nonnumeric domains (such as colors), you must map the values in the domain to integers. For more information, see "Example 6.5: Car Painting Problem" on page 231.

You can use the CLP solver to find one or more (and in some instances, all) solutions to a CSP that has linear, logical, and global constraints. The numeric components of all variable domains are required to be integers.

Getting Started: CLP Solver

The following examples illustrate the use of the CLP solver in formulating and solving two well-known logical puzzles in constraint programming.

Send More Money

The Send More Money problem consists of finding unique digits for the letters D, E, M, N, O, R, S, and Y such that S and M are different from zero (no leading zeros) and the following equation is satisfied:

$$\begin{array}{r} \text{SEND} \\ +\text{MORE} \\ \hline \text{MONEY} \end{array}$$

You can use the CLP solver to formulate this problem as a CSP by using an integer variable to represent each letter in the expression. The comments before each statement in the following code introduce variables, domains, and constraints:

```
/*  Send More Money  */

proc optmodel;
   /* Declare all variables as integer. */
   var S integer, E integer, N integer, D integer, M integer, O integer,
       R integer, Y integer;
   /* Set all domains to between 0 and 9. Domains are unbounded by default.
      Always declare domains to be as tight as possible. */
   for {j in 1.._NVAR_} do;
      _VAR_[j].lb = 0;
      _VAR_[j].ub = 9;
   end;
   /* Describe the arithmetic constraint.*/
   con Arithmetic:           1000*S + 100*E + 10*N + D
                  +          1000*M + 100*O + 10*R + E
                  = 10000*M + 1000*O + 100*N + 10*E + Y;
   /* Forbid leading letters from taking the value zero.
      Constraint names are optional. */
   con S ne 0;
   con M ne 0;
   /* Require all variables to take distinct values. */
   con alldiff(S E N D M O R Y);

   solve;
   print S E N D M O R Y;
quit;
```

The domain of each variable is the set of digits 0 through 9. The VAR statement identifies the variables in the problem. The Arithmetic constraint defines the linear constraint SEND + MORE = MONEY and the restrictions that S and M cannot take the value 0. (Alternatively, you can simply specify the domain for S and M as $\geq 1 \leq 9$ in the VAR statement.) Finally, the ALLDIFF predicate enforces the condition that the assignment of digits should be unique.

The PRINT statement produces the solution output as shown in Figure 6.1.

Figure 6.1 Solution to SEND + MORE = MONEY

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	(0)
Objective Type	Constant
Number of Variables	8
Bounded Above	0
Bounded Below	0
Bounded Below and Above	8
Free	0
Fixed	0
Binary	0
Integer	8
Number of Constraints	4
Linear LE (<=)	0
Linear EQ (=)	1
Linear GE (>=)	0
Linear LT (<)	0
Linear NE (~=)	2
Linear GT (>)	0
Linear Range	0
Alldiff	1
Element	0
GCC	0
Lexico (<=)	0
Lexico (<)	0
Pack	0
Reify	0
Constraint Coefficients	10

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Figure 6.1 *continued*

Solution Summary	
Solver	CLP
Variable Selection	MINR
Objective Function	(0)
Solution Status	Solution Limit Reached
Objective Value	0
Solutions Found	1
Presolve Time	0.00
Solution Time	0.02

S	E	N	D	M	O	R	Y
9	5	6	7	1	0	8	2

The CLP solver determines the following unique solution to this problem:

$$\begin{array}{r} 9\,5\,6\,7 \\ +\,1\,0\,8\,5 \\ \hline 1\,0\,6\,5\,2 \end{array}$$

Eight Queens

The Eight Queens problem is a special instance of the N-Queens problem, where the objective is to position N queens on an $N \times N$ chessboard such that no two queens can attack each other. The CLP solver provides an expressive constraint for variable arrays that can be used for solving this problem very efficiently.

You can model this problem by using a variable array $\boldsymbol{a}$ of dimension N, where a_i is the row number of the queen in column i. Because no two queens can be in the same row, it follows that all the a_i's must be pairwise distinct.

In order to ensure that no two queens can be on the same diagonal, the following two expressions should be true for all i and j:

$$a_j - a_i <> j - i$$

$$a_j - a_i <> i - j$$

These expressions can be reformulated as follows:

$$a_i - i <> a_j - j$$

$$a_i + i <> a_j + j$$

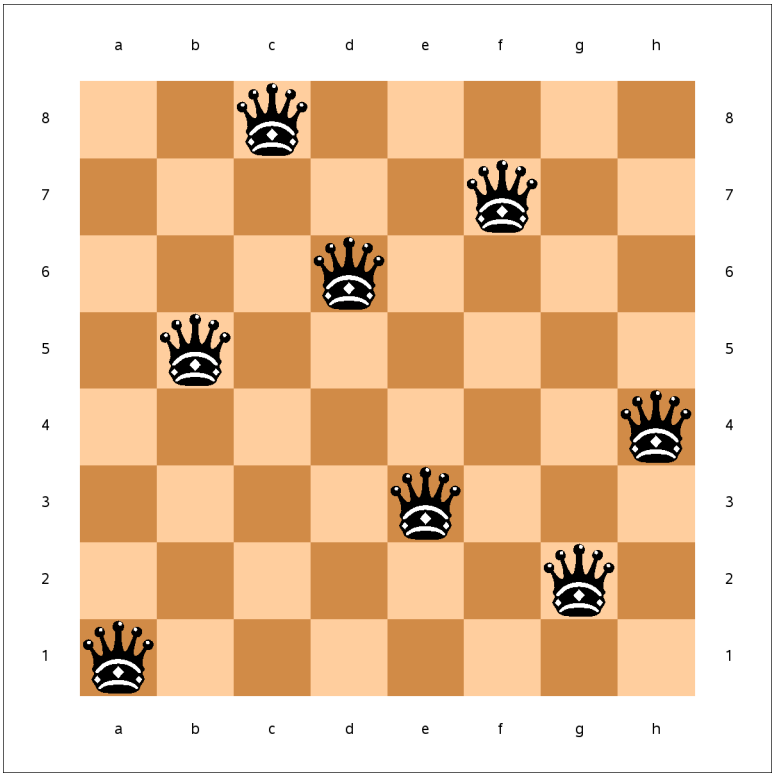
Hence, the $(a_i + i)$'s are pairwise distinct, and the $(a_i - i)$'s are pairwise distinct. One possible formulation is as follows:

```
proc optmodel;
   num n init 8;
   var A {1..n}        >= 1     <= n     integer;
   /* Define artificial offset variables. */
   var B {1..n, -1..1} >= 1 - n <= n + n integer;
   con Bdef {i in 1..n, k in -1..1}:
      B[i,k] = A[i] + k * i;

   con OffsetsMustBeAlldifferent {k in -1..1}:
      alldiff({i in 1..n} B[i,k]);
   solve with CLP / varselect=fifo;
   /* Replicate typical PROC CLP output from an OPTMODEL array */
   create data out from {i in 1..n}<col('A'||i)=A[i]>;
quit;
```

The VARSELECT= option specifies the variable selection strategy to be first-in, first-out—the order in which the CLP solver encounters the variables.

The corresponding solution to the Eight Queens problem is displayed in Figure 6.2.

Figure 6.2 A Solution to the Eight Queens Problem



Syntax: CLP Solver

SOLVE WITH CLP /
< **FINDALLSOLNS** >
< **MAXSOLNS=***number* >
< **MAXTIME=***number* >
< **NOPREPROCESS** >
< **OBJTOL=***number* >
< **PREPROCESS** >
< **SHOWPROGRESS** >
< **TIMETYPE=***number* | *string* >
< **VARASSIGN=***string* >
< **VARSELECT=***string* >
;

The section "Functional Summary" on page 199 provides a quick reference for each option. Each option is then described in more detail in its own section, in alphabetical order.

Functional Summary

Table 6.1 summarizes the options available in the SOLVE WITH CLP statement.

Table 6.1 Functional Summary of SOLVE WITH CLP Options

Description	Option
General Options	
Finds all possible solutions	FINDALLSOLNS
Specifies the number of solution attempts	MAXSOLNS=
Specifies the maximum time to spend calculating results	MAXTIME=
Suppresses preprocessing	NOPREPROCESS
Specifies the tolerance of the objective value	OBJTOL=
Permits preprocessing	PREPROCESS
Indicates progress in the log	SHOWPROGRESS
Specifies whether time units are CPU time or real time	TIMETYPE=
Specifies the variable assignment strategy	VARASSIGN=
Specifies the variable selection strategy	VARSELECT=

SOLVE WITH CLP Statement

SOLVE WITH CLP / < *options* > ;

The SOLVE WITH CLP statement invokes the CLP solver. You can specify the following *options* to define various processing and diagnostic controls and to tune the algorithm to run.

General Options

You can specify the following general options.

FINDALLSOLNS
ALLSOLNS
FINDALL
attempts to find all possible solutions to the CSP.

MAXSOLNS=*number*
specifies the number of solution attempts to be generated for the CSP. By default, MAXSOLNS=1.

MAXTIME=*number*
specifies the maximum time to spend calculating results. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

NOPREPROCESS
suppresses any preprocessing that would usually be performed for the problem.

OBJTOL=*number*
specifies the tolerance of the objective value. By default, OBJTOL=1E–6.

PREPROCESS
permits any preprocessing that would usually be performed for the problem.

SHOWPROGRESS
prints a message to the log whenever a solution is found.

TIMETYPE=*number* | *string*
specifies whether to use CPU time or real time for the MAXTIME= option. Table 6.2 describes the valid values of the TIMETYPE= option.

Table 6.2 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

By default, TIMETYPE=REAL if the CLP solver is invoked in a PROC OPTMODEL COFOR loop; otherwise, the default is TIMETYPE=CPU.

VARASSIGN=*string*
specifies the variable assignment strategy. You can specify two value selection strategies:

- MAX, which selects the maximum value from the domain of the selected variable
- MIN, which selects the minimum value from the domain of the selected variable

By default, VARASSIGN=MIN.

VARSELECT=*string*

specifies the variable selection strategy. The strategy could be static, dynamic, or conflict-directed. Typically, static strategies exploit information about the initial state of the search, whereas dynamic strategies exploit information about the current state of the search process. Conflict-directed strategies exploit information from previous states of the search process as well as the current state (Boussemart et al. 2004). Table 6.3 describes the valid values of the VARSELECT= option.

Table 6.3 Values for VARSELECT= Option

string	**Description**
Static strategies	
FIFO	Uses the first-in, first-out ordering of the variables as encountered by the procedure after adjusting for the values in the .priority suffix
MAXCS	Selects the variable that has the maximum number of constraints
Dynamic strategies	
DOMDDEG	Selects the variable that has the smallest ratio of domain size to dynamic degree
DOMDEG	Selects the variable that has the smallest ratio of domain size to degree
MAXC	Selects the variable that has the largest number of active constraints
MINR	Selects the variable that has the smallest range (that is, the minimum value of the upper bound minus the lower bound)
MINRMAXC	Selects the variable that has the smallest range, breaking ties by selecting one that has the largest number of active constraints
Conflict-directed strategies	
DOMWDEG	Selects the variable that has the smallest ratio of domain size to weighted degree
WDEG	Selects the variable that has the largest weighted degree

The dynamic strategies embody the "fail-first principle" (FFP) of Haralick and Elliott (1980), which suggests, "To succeed, try first where you are most likely to fail." By default, VARSELECT=MINR.

You can change the order in which the CLP solver selects variables after you have declared them by setting the .priority suffix for your variables. Variables with higher priority are selected first.

PROC OPTMODEL usually honors the variable declaration order when it invokes the CLP solver. However, when variables are indexed over dynamically computed sets, the variable order might be undefined. The value of .priority always overrides declaration order. To ensure that the CLP solver considers variables in the exact order that you want, set the .priority suffix for all variables. You can also experiment with the priorities by changing the suffixes without having to reorder the variable declarations in your code.

For more information, see the section "Integer Variable Suffixes" on page 137 in Chapter 5, "The OPTMODEL Procedure."

Predicates

A predicate asserts a fact about its subject. You use a predicate as the first identifier in a constraint declaration to define what must hold true in any feasible solution. The following is an example of a constraint that uses the ELEMENT predicate:

```
var X integer, Y integer;
con ElementConstraintExample:   element(X, 1 2 3 4, Y);
```

The following predicates are available in PROC OPTMODEL:

ALLDIFF(*variable-list***)**
ELEMENT(*scalar-variable***,***data-list***,***scalar-variable***)**
GCC(*variable-list***,***set-of-numeric-triplets***)**
LEXICO(*variable-list order-type variable-list***)**
PACK(*variable-list***,***data-list***,***variable-list***)**
REIFY(*scalar-variable***,** *linear-constraint***)**

Common Syntax Components

The following syntax components are used in multiple predicates. They depend on the definition of an *identifier-expression*. For more information, see the section "Identifier Expressions" on page 102 in Chapter 5, "The OPTMODEL Procedure." For information about other syntactic components, see the section that describes the corresponding predicate.

data-list

is a space-separated list of items, each of which can be prefixed by an indexing set and conforms to one of the following syntaxes:

- a number
- an *identifier-expression*, but excluding suffixes
- the name of a numeric array
- (*expression*), which must evaluate to a number

For example, the first three constraints in the following statements refer to valid *data-lists*, whereas the last four do not. Each incorrect constraint is preceded by a comment that explains why the constraint is incorrect.

```
var X integer, Y integer, Z integer;
num n;
con CorrectDataList1:   element(X, 1 2 3 4, Y);
con CorrectDataList2:   element(X, 1 2 3 4 n, Y);
con CorrectDataList3:   element(X, {i in 1..4} i n, Y);
/* The parenthesis imply a scalar expression */
con IncorrectDataList1: element(X, ({i in 1..4} i) n, Y);
/* [1 2 3 4] is an array initializer, not an array name */
con IncorrectDataList2: element(X, [1 2 3 4], Y);
/* Z is a variable */
con IncorrectDataList3: element(X, 1 2 3 4 Z, Y);
/* Impossible to distinguish Z.sol from [ Z . sol ] */
con IncorrectDataList4: element(X, 1 2 3 4 Z.sol, Y);
```

scalar-variable

is an *identifier-expression* that refers to a single variable (that is, not to an array of variables).

variable-list

is a space-separated list of *identifier-expressions*, each of which can be prefixed by an indexing set and must resolve to a variable or a variable array. For example, the first three constraints in the following statements refer to valid *variable-lists*, whereas the last four do not. Each incorrect constraint is preceded by a comment that explains why the constraint is incorrect.

```
var X{1..3} integer, A integer, B integer;
con CorrectVariableList1: alldiff({j in 1..3} X[j]);
con CorrectVariableList2: alldiff(A B);
con CorrectVariableList3: alldiff({j in 1..3} X[j] A B);
/* Indexing is not distributive in variable lists */
var Y{1..3} integer;
con IncorrectVariableList1: alldiff({j in 1..3} (X[j] Y[j]));
/* literals or expressions are not allowed in variable lists */
con IncorrectVariableList2: alldiff(1 A B);
/* literals or expressions are not allowed in variable lists */
con IncorrectVariableList3: alldiff(A.dual B);
/* you must refer only to variables */
num n;
con IncorrectVariableList4: alldiff(A B n);
```

ALLDIFF Predicate

ALLDIFF(*variable-list***)**
ALLDIFFERENT(*variable-list***)**

The ALLDIFF predicate defines an all-different constraint, which defines a unique global constraint on a set of variables that requires all of them to be different from each other. A global constraint is equivalent to a conjunction of elementary constraints.

The syntax of the all-different constraint consists of one part, a *variable-list*, which is defined in the section "Common Syntax Components" on page 202. For example, the statements

```
var X{1..3} integer, A integer, B integer;
con AD1: alldiff({j in 1..3} X[j]);
con AD2: alldiff(A B);
```

are equivalent to

$$\begin{aligned} X[1] &\neq X[2] \text{ AND} \\ X[2] &\neq X[3] \text{ AND} \\ X[1] &\neq X[3] \text{ AND} \\ A &\neq B \end{aligned}$$

To apply the all-different constraint to all the variables, use the problem symbol _VAR_ as follows:

```
con alldiff(_VAR_);
```

For a description of problem symbols, see the section "Problem Symbols" on page 152 in Chapter 5, "The OPTMODEL Procedure".

ELEMENT Predicate

ELEMENT(*scalar-variable***,***data-list***,***scalar-variable***)**

The ELEMENT predicate specifies an array element lookup constraint, which enables you to define dependencies (which are not necessarily functional) between variables.

The predicate ELEMENT(I, L, V) sets the variable V to be equal to the Ith element in the list L, where $L = (v_1, \ldots, v_n)$ is a list of values (not necessarily distinct) that the variable V can take. The variable I is the index variable, and its domain is considered to be $[1, n]$. Each time the domain of I is modified, the domain of V is updated, and vice versa.

For example, the following statements use the ELEMENT predicate to determine whether there are squares greater than 1 that are also elements of the Fibonacci sequence:

```
/* Are there any squares > 1 in the Fibonacci sequence? */
proc optmodel;
   num n = 20;
   /* 1 appears twice in the Fibonacci sequence */
   num fib{i in 1..n} = if i < 3 then 1 else fib[i-1] + fib[i-2];

   var IFib integer, ISq integer,
       XFib integer, XSq integer;

   con IsFibAndIsSquare: XFib = XSq;
   /* You can use a numeric array to refer to a list */
   con IdxOfFib: element( IFib, fib, XFib );
   /* You can also build a list from a set iterator */
   con IdxOfSq:  element( ISq,  {i in 2..n} (i * i),  XSq );
   solve;
   print XFib XSq;
quit;
```

An element constraint enforces the propagation rules

$$V = v \Leftrightarrow I \in \{i_1, \ldots, i_m\}$$

where v is a value in the list L and $i_1, \ldots, i_m$ are all the indices in L whose value is v.

An element constraint is equivalent to a conjunction of reify and linear constraints. For example, both of the following examples implement the quadratic function, $Y = X^2$:

- Using the ELEMENT predicate:

```
proc optmodel;
   var X >= 1 <= 5 integer, Y >= 1 <= 25 integer;
   num a {i in 1..5} = i^2;
   con Mycon: element(X, a, Y);
   solve;
quit;
```

- Using linear constraints and the REIFY predicate:

```
proc optmodel;
   var X >= 1 <= 5 integer, Y >= 1 <= 25 integer, R {1..5} binary;
   con MyconX {i in 1..5}:
      reify(R[i], X = i);
   con MyconY {i in 1..5}:
      reify(R[i], Y = i^2);
   con SumToOne:
      sum {i in 1..5} R[i] = 1;
   solve;
quit;
```

You can also use element constraints to define positional mappings between two variables. For example, suppose the function $Y = X^2$ is defined on only odd numbers in the interval $[-5, 5]$. You can relate X and Y by using two element constraints and an artificial index variable:

```
var I integer, X integer, Y integer;
/* You can also build a list by providing explicit literals. */
con XsubI: element (I, -5 -3 -1 1 3  5, X);
con YsubI: element (I, 25  9  1 1 9 25, Y);
```

GCC Predicate

GCC(*variable-list***,***set-of-numeric-triplets***)**

The GCC predicate specifies a global cardinality constraint (GCC), which sets the minimum and maximum number of times each value can be assigned to a group of variables.

The syntax of the GCC constraint consists of two parts:

variable-list
: See the section "Common Syntax Components" on page 202.

set-of-numeric-triplets
: The triplets $< v, l_v, u_v >$ provide, for each value v, the minimum l_v and maximum u_v number of times that v can be assigned to the variables in the *variable-list*. The PROC OPTMODEL option INTFUZZ= determines which values are rounded to integers.

Consider the following statements:

```
var X {1..6} >= 1 <= 4 integer;
con Mycon: gcc(X, /<1,1,2>, <2,1,3>, <3,1,3>, <4,2,3>/);
```

These statements specify a constraint that expresses the following requirements about the values of variables $\{X[1], \ldots, X[6]\}$:

- The value 1 must appear at least once but no more than twice.
- The value 2 must appear at least once but no more than three times.
- The value 3 must appear at least once but no more than three times.
- The value 4 must appear at least twice but no more than three times.

The assignment $X[1] = 1, X[2] = 1, X[3] = 2, X[4] = 3, X[5] = 4$, and $X[6] = 4$ satisfies the constraint.

In general, a GCC constraint consists of a set of variables $\{x_1, \ldots, x_n\}$ and, for each value v in $D = \bigcup_{i=1,\ldots,n} \mathrm{Dom}(x_i)$, a pair of numbers l_v and u_v. A GCC is satisfied if and only if the number of times that a value v in D is assigned to the variables $x_1, \ldots, x_n$ is at least l_v and at most u_v.

Values in the domain of *variable-list* that do not appear in any triplet are unconstrained. They can be assigned to as many of the variables in *variable-list* as needed to produce a feasible solution.

The following statements specify that each of the values in the set $\{1, \ldots, 9\}$ can be assigned to at most one of the variables $X[1], \ldots, X[9]$:

```
var X {1..9} >= 1 <= 9 integer;
con Mycon: gcc(X, setof{i in 1..9} <i,0,1>);
```

Note that the preceding global cardinality constraint is equivalent to the all-different constraint that is expressed as follows:

```
var X {1..9} >= 1 <= 9 integer;
con Mycon: alldiff(X);
```

The global cardinality constraint also provides a convenient way to define disjoint domains for a set of variables. For example, the following syntax limits assignment of the variables $X[1], \ldots, X[9]$ to even numbers between 0 and 10:

```
var X {1..9} >= 0 <= 10 integer;
con Mycon: gcc(X, setof{i in 1..9 by 2} <i,0,0>);
```

LEXICO Predicate

LEXICO(*variable-list order-type variable-list*)

The LEXICO predicate defines a lexicographic ordering constraint, which compares two arrays of the same size from left to right. For example, a standings table in a sports competition is usually ordered lexicographically, with certain attributes (such as wins or points) to the left of others (such as goal difference).

The *order-type* can be either $\leq$, to indicate lexicographically less than or equal to ($\leq_{\text{lex}}$), or $<$, to indicate lexicographically less than ($<_{\text{lex}}$). Given two n-tuples $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, the n-tuple x is lexicographically less than or equal to y ($x \leq_{\text{lex}} y$) if and only if

$$\left(x_i = y_i \ \forall i = 1, \ldots, n\right) \vee \left(\exists j \text{ with } 1 \leq j \leq n \text{ s.t. } x_i = y_i \ \forall i = 1, \ldots, j-1 \text{ and } x_j < y_j\right)$$

The n-tuple x is lexicographically less than y ($x <_{\text{lex}} y$) if and only if $x \leq_{\text{lex}} y$ and $x \neq y$. Equivalently, $x <_{\text{lex}} y$ if and only if

$$\exists j \text{ with } 1 \leq j \leq n \text{ s.t. } x_i = y_i \ \forall i = 1, \ldots, j-1 \text{ and } x_j < y_j$$

Informally, you can think of the lexicographic constraint $\leq_{\text{lex}}$ as sorting the n-tuples in alphabetical order. Mathematically, $\leq_{\text{lex}}$ is a partial order on a subset of n-tuples, and $<_{\text{lex}}$ is a strict partial order on a subset of n-tuples (Brualdi 2010).

For example, you can express the lexicographic constraint $(X[1], \ldots, X[6]) \leq_{\text{lex}} (Y[1], \ldots, Y[6])$ by using a LEXICO predicate as follows:

```
con Mycon: lexico({j in 1..6} X[j] <= {j in 1..6} Y[j]);
```

The assignment $X[1]=1,\ X[2]=2,\ X[3]=2,\ X[4]=1,\ X[5]=2,\ X[6]=5,\ Y[1]=1,\ Y[2]=2,\ Y[3]=2,\ Y[4]=1,\ Y[5]=4$, and $Y[6]=3$ satisfies this constraint because $X[i] = Y[i]$ for $i = 1,\ldots,4$ and $X[5] < Y[5]$. The fact that $X[6] > Y[6]$ is irrelevant in this ordering.

Lexicographic ordering constraints can be useful for breaking a certain type of symmetry that arises in CSPs and involves matrices of decision variables. Frisch et al. (2002) introduce an optimal algorithm to establish generalized arc consistency (GAC) for the $\leq_{\text{lex}}$ constraint between two vectors of variables.

PACK Predicate

PACK(*variable-list***,***data-list***,***variable-list***)**

The PACK predicate specifies a pack constraint, which is used to assign items to bins, subject to the sizes of the items and the capacities of the bins.

For example, suppose you have three bins, whose capacities are 3, 4, and 5, and you have five items of sizes 4, 3, 2, 2, and 1, to be assigned to these three bins. The following statements formulate the problem and find a solution:

```
proc optmodel;
   var SpaceUsed {bin in 1..3} integer >= 0 <= bin + 2;
   var WhichBin {1..5} >= 1 <= 3 integer;
   num itemSize {1..5} = [4 3 2 2 1];
   con pack( WhichBin, itemSize, SpaceUsed );
   solve / findall;
quit;
```

Each row of Table 6.4 represents a solution to the problem. The number in each item column is the number of the bin to which the corresponding item is assigned.

Table 6.4 Bin Packing Solutions

WhichBin Variable				
WhichBin[1]	**WhichBin[2]**	**WhichBin[3]**	**WhichBin[4]**	**WhichBin[5]**
2	3	3	1	1
2	3	1	3	1
2	1	3	3	3
3	1	2	2	3

When you assign a set of k items to m bins, the item variable $b_i, i \in 1,\ldots,k$ (WhichBin in the preceding example) contains the bin number for the ith item. The constant s_i (itemSize in the preceding example) holds the size or weight of the ith item. The domain of the load variable l_j (SpaceUsed in the preceding example) constrains the capacity of bin j. The value of l_j in the solution is the amount of space used in bin j.

NOTE: It can be more efficient to assign higher priority to item variables than to load variables, and within the item variables, to assign higher priority to larger itmes.

REIFY Predicate

REIFY(*scalar-variable*, *linear-constraint*)

The REIFY predicate specifies a reify constraint, which associates a binary variable with a constraint. The value of the binary variable is 1 or 0, depending on whether the constraint is satisfied or not, respectively. The constraint is said to be reified, and the binary variable is referred to as the *control variable*. Currently, only linear constraints can be reified.

The REIFY predicate provides a convenient mechanism for expressing logical constraints, such as disjunctive and implicative constraints. For example, the disjunctive constraint

$$(3X + 4Y < 20) \vee (5X - 2Y > 50)$$

can be expressed by the following statements:

```
var X integer, Y integer,
    P binary,  Q binary;
con reify(P, 3 * X + 4 * Y < 20);
con reify(Q, 5 * X - 2 * Y > 50);
con AtLeastOneHolds: P + Q >= 1;
```

The binary variable P reifies the linear constraint

$$3X + 4Y < 20$$

The binary variable Q reifies the linear constraint

$$5X - 2Y > 50$$

The following linear constraint enforces the desired disjunction:

$$P + Q \geq 1$$

The following implicative constraint

$$(3X + 4Y < 20) \Rightarrow (5X - 2Y > 50)$$

can be enforced by the linear constraint

$$P \leq Q$$

You can also use the REIFY constraint to express a constraint that involves the absolute value of a variable. For example, the constraint

$$|X| = 5$$

can be expressed by the following statements:

```
var X integer, P binary, Q binary;
con Xis5:      reify(P, X = 5);
con XisMinus5: reify(Q, X = -5);
con OneMustHold: P + Q = 1;
```

Details: CLP Solver

Types of CSPs

The CLP solver is a finite-domain constraint programming solver for CSPs. A *standard CSP* is characterized by integer variables, linear constraints, global constraints, and reify constraints. The solver expects only linear, ALLDIFF, ELEMENT, GCC, LEXICO, PACK, and REIFY predicates.

Both PROC OPTMODEL and PROC CLP support standard CSPs. The CLP procedure also supports *scheduling CSP*s, which are characterized by activities, temporal constraints, and resource requirement constraints. For more information about the CLP procedure see *SAS/OR User's Guide: Constraint Programming*.

Techniques for Solving CSPs

Several techniques for solving CSPs are available. Kumar (1992) and Tsang (1993) present a good overview of these techniques. It should be noted that the satisfiability problem (SAT) (Garey and Johnson 1979) can be regarded as a CSP. Consequently, most problems in this class are nondeterministic polynomial-time complete (NP-complete) problems, and a backtracking search mechanism is an important technique for solving them (Floyd 1967).

One of the most popular tree search mechanisms is chronological backtracking. However, a chronological backtracking approach is not very efficient because conflicts are detected late; that is, the approach is oriented toward *recovering* from failures rather than *avoiding* them to begin with. The search space is reduced only after a failure is detected, and the performance of this technique is drastically reduced as the problem size increases. Another drawback of using chronological backtracking, for the same reason, is encountering repeated failures, sometimes called "thrashing." The presence of late detection and "thrashing" has led researchers to develop consistency techniques that can achieve superior pruning of the search tree. This strategy uses constraints actively, rather than passively.

Constraint Propagation

A more efficient technique than backtracking is constraint propagation, which uses consistency techniques to effectively prune the domains of variables. Consistency techniques are based on the idea of a priori pruning, which uses the constraint to reduce the domains of the variables. Consistency techniques are also known as relaxation algorithms (Tsang 1993), and the process is also called problem reduction, domain filtering, or pruning.

One of the earliest applications of consistency techniques was in the AI field to solve the scene labeling problem, which required recognizing objects in three-dimensional space by interpreting two-dimensional line drawings of the object. The Waltz filtering algorithm (Waltz 1975) analyzes line drawings by systematically labeling the edges and junctions while maintaining consistency between the labels.

Constraint propagation is characterized by the extent of propagation (also called the level of consistency) and whether domain propagation or interval propagation is the domain pruning scheme that is followed. In practice, interval propagation is preferred because of its lower computational costs. This mechanism is

discussed in detail in Van Hentenryck (1989). However, constraint propagation is not a complete solution technique and needs to be complemented by a search technique in order to ensure success (Kumar 1992).

Finite-Domain Constraint Programming

Finite-domain constraint programming is an effective and complete solution technique that embeds incomplete constraint propagation techniques into a nondeterministic backtracking search mechanism that is implemented as follows: Whenever a node is visited, constraints are propagated to attain a desired level of consistency. If the domain of each variable reduces to a singleton set, the node represents a solution to the CSP. If the domain of a variable becomes empty, the node is pruned. Otherwise a variable is selected, its domain is distributed, and a new set of CSPs is generated, each of which is a child node of the current node. Several factors play a role in determining the outcome of this mechanism, such as the extent of propagation (or level of consistency enforced), the variable selection strategy, and the variable assignment or domain distribution strategy.

For example, the lack of any propagation reduces this technique to a simple generate-and-test approach, whereas performing consistency checking using variables that are already selected reduces this approach to chronological backtracking, one of the systematic search techniques. These are also known as look-back schemas, because they share the disadvantage of late conflict detection. Look-ahead schemas, on the other hand, work to prevent future conflicts. Some popular examples of look-ahead schemas, in increasing degree of consistency level, are forward checking (FC), partial look ahead (PLA), and full look ahead (LA) (Kumar 1992). Forward checking enforces consistency between the current variable and future variables; PLA and LA extend this even further to pairs of not yet instantiated variables.

Two important consequences of this technique are that inconsistencies are discovered early and that the current set of alternatives that are coherent with the existing partial solution is dynamically maintained. These consequences are powerful enough to prune large parts of the search tree, thereby reducing the "combinatorial explosion" of the search process. However, although constraint propagation at each node results in fewer nodes in the search tree, the processing at each node is more expensive. The ideal scenario is to strike a balance between the extent of propagation and the subsequent computation cost.

Variable selection is another strategy that can affect the solution process. The order in which variables are chosen for instantiation can have a substantial impact on the complexity of the backtrack search. Several heuristics have been developed and analyzed for selecting variable ordering. One of the most common ones is a dynamic heuristic based on the *fail-first principle* (Haralick and Elliott 1980), which selects the variable whose domain has minimal size. Subsequent analysis of this heuristic by several researchers has validated this strategy as providing substantial improvement for a significant class of problems. Another popular strategy is to instantiate the most constrained variable first. Both these strategies are based on the principle of selecting the variable most likely to fail and detecting such failures as early as possible.

The domain distribution strategy for a selected variable is yet another area that can influence the performance of a backtracking search. However, good value-ordering heuristics are expected to be very problem-specific (Kumar 1992).

Consistency Techniques

The CLP solver features a full look-ahead algorithm for standard CSPs that follows a strategy of maintaining a version of generalized arc consistency that is based on the AC-3 consistency routine (Mackworth 1977). This strategy maintains consistency between the selected variables and the unassigned variables and also maintains consistency between unassigned variables.

Selection Strategy

A search algorithm for CSPs searches systematically through the possible assignments of values to variables. The order in which a variable is selected can be based on a *static* ordering, which is determined before the search begins, or on a *dynamic* ordering, in which the choice of the next variable depends on the current state of the search. The VARSELECT= option in the SOLVE statement defines the variable selection strategy for a standard CSP. The default strategy is the dynamic MINR strategy, which selects the variable that has the smallest range.

Assignment Strategy

After a variable is selected, the assignment strategy dictates the value that is assigned to it. For variables, the assignment strategy is specified in the VARASSIGN= option in the SOLVE statement. The default assignment strategy selects the minimum value from the domain of the selected variable.

Differences between PROC OPTMODEL and PROC CLP

You can invoke the CLP solver from PROC OPTMODEL by using any of the predicates that are defined in the standard mode of PROC CLP. The *standard mode* gives you access to all-different, element, GCC, linear, pack, and reify constraints.

To replicate the FOREACH statement that PROC CLP supports, you can use PROC OPTMODEL's expressions and iteration machinery. For an example, see the Eight Queens example in the "Getting Started: CLP Solver" on page 194. For more information about the FOREACH predicate, see *SAS/OR User's Guide: Constraint Programming*.

In addition to the predicates that are defined in this chapter, PROC CLP provides several constraints and capabilities that simplify the modeling of scheduling-oriented CSPs. For more information about those statements, see the section "Details: CLP Procedure" (Chapter 3, *SAS/OR User's Guide: Constraint Programming*).

PROC OPTMODEL has different syntax and semantics for variable declarations:

- Because all CLP variables are discrete, you must declare every variable that a CLP model uses as integer or binary.
- The default variable bounds in PROC OPTMODEL are $-\infty$ to ∞. The default lower bound in PROC CLP is 0. Thus, to replicate the behavior of PROC CLP, you must explicitly add a lower bound of 0 to the variable declaration.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro variable called _OROPTMODEL_, which contains a character string. The variable contains information about the execution of the most recently invoked solver.

Each keyword and value pair in _OROPTMODEL_ also appears in two other places: the PROC OPTMODEL automatic arrays _OROPTMODEL_NUM_ and _OROPTMODEL_STR_; and the ODS tables ProblemSummary and SolutionSummary, which appear after a SOLVE statement, unless you set the PRINTLEVEL= option to NONE. You can use these variables to obtain details about the solution.

After the solver is called, the various keywords in the variable are interpreted as follows.

STATUS
indicates the solver status at termination. It can take one of the following values:

OK	The solver terminated normally.
SYNTAX_ERROR	The use of syntax is incorrect.
DATA_ERROR	The input data are inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem in reading or writing of data has occurred.
SEMANTIC_ERROR	An evaluation error, such as an invalid operand type, has occurred.
ERROR	The status cannot be classified into any of the preceding categories.

SOLUTION_STATUS
indicates the solution status at termination. It can take one of the following values:

ABORT_NOSOL	The solver was stopped by the user and did not find a solution.
ABORT_SOL	The solver was stopped by the user but still found a solution.
ALL_SOLUTIONS	All solutions were found.
BAD_PROBLEM_TYPE	The problem type is not supported by the solver.
CONDITIONAL_OPTIMAL	The optimality of the solution cannot be proven.
ERROR	The algorithm encountered an error.
FAIL_NOSOL	The solver stopped because of errors and did not find a solution.
FAIL_SOL	The solver stopped because of errors but still found a solution.
INFEASIBLE	The problem is infeasible.
INTERRUPTED	The solver was interrupted by the system or the user before completing its work.

OK	The algorithm terminated normally.
OPTIMAL	The solution is optimal.
OUTMEM_NOSOL	The solver ran out of memory and either did not find a solution or failed to output the solution because of insufficient memory.
OUTMEM_SOL	The solver ran out of memory but still found a solution.
SOLUTION_LIM	The solver reached the maximum number of solutions specified in the MAXSOLNS= option.
TIME_LIM_NOSOL	The solver reached the execution time limit specified in the MAXTIME= option and did not find a solution.
TIME_LIM_SOL	The solver reached the execution time limit specified in the MAXTIME= option and found a solution.
UNBOUNDED	The problem is unbounded.

ALGORITHM
indicates the algorithm that produces the solution data in the macro variable. This term appears only when STATUS=OK. It can take only one value in the CLP solver: CLP, which indicates that the constraint satisfaction algorithm produced the solution data.

OBJECTIVE
indicates the objective value that the solver obtained at termination. If a problem does not have an explicit objective, the value of this keyword in the _OROPTMODEL_ macro variable is missing (.).

PRESOLVE_TIME
indicates the time (in seconds) that the algorithm used for preprocessing. You can use the TIMETYPE= option to select real time or CPU time.

SOLUTION_TIME
indicates the time (in seconds) that the algorithm used to perform iterations to solve the problem. You can use the TIMETYPE= option to select real time or CPU time.

SOLUTIONS_FOUND
indicates the number of solutions found, which might be 0 if the problem is infeasible. This keyword is always present in the solution status when you call the CLP solver. The value might not be the total number of solutions possible (for example, if the solver reached its time limit).

Examples: CLP Solver

Example 6.1: Logic-Based Puzzles

Many logic-based puzzles can be formulated as CSPs. Several such puzzles are shown in this example.

Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle that uses a partially filled 9 × 9 grid. The objective is to fill the grid with the digits 1 to 9 so that each column, each row, and each of the nine 3 × 3 blocks contains only one of each digit. Figure 6.1.1 shows an example of a sudoku grid.

Output 6.1.1 Example of an Unsolved Sudoku Grid

		5			7			1
	7			9			3	
			6					
		3			1			5
	9			8			2	
1			2			4		
		2			6			9
				4			8	
8			1			5		

This example illustrates the use of the all-different constraint to solve the preceding sudoku problem. The data set Indata contains the partially filled values for the grid.

```
data Indata;
   input C1-C9;
   datalines;
. . 5 . . 7 . . 1
. 7 . . 9 . . 3 .
. . . 6 . . . . .
. . 3 . . 1 . . 5
. 9 . . 8 . . 2 .
1 . . 2 . . 4 . .
. . 2 . . 6 . . 9
. . . . 4 . . 8 .
8 . . 1 . . 5 . .
;
```

Let the variable $X_{ij} (i = 1, \ldots, 9, j = 1, \ldots, 9)$ represent the value of cell (i, j) in the grid. The domain of each of these variables is $[1, 9]$. When cell (i, j) is not missing in the data set, X_{ij} is fixed to that value.

Three sets of all-different constraints specify the required rules for each row, each column, and each of the 3×3 blocks. The RowCon constraint forces all values in row i to be different, the ColumnCon constraint forces all values in column j to be different, and the BlockCon constraint forces all values in each block to be different.

The following statements express the preceding constraints in PROC OPTMODEL and solve the sudoku puzzle:

```
proc optmodel;
   /* Declare variables */
   set ROWS = 1..9;
   set COLS = ROWS; /* Use an alias for convenience and clarity */
   var X {ROWS, COLS} >= 1 <= 9 integer;

   /* Nine row constraints */
   con RowCon {i in ROWS}:
      alldiff({j in COLS} X[i,j]);

   /* Nine column constraints */
   con ColCon {j in COLS}:
      alldiff({i in ROWS} X[i,j]);

   /* Nine 3x3 block constraints */
   con BlockCon {s in 0..2, t in 0..2}:
      alldiff({i in 3*s+1..3*s+3, j in 3*t+1..3*t+3} X[i,j]);

   /* Fix variables to cell values */
   /* X[i,j] = c[i,j] if c[i,j] is not missing */
   num c {ROWS, COLS};
   read data indata into [_N_] {j in COLS} <c[_N_,j]=col('C'||j)>;
   for {i in ROWS, j in COLS: c[i,j] ne .}
      fix X[i,j] = c[i,j];

   solve;
quit;
```

Output 6.1.2 shows the solution.

Output 6.1.2 Solution of the Sudoku Grid

9	8	5	3	2	7	6	4	1
6	7	1	5	9	4	2	3	8
3	2	4	6	1	8	9	5	7
2	4	3	7	6	1	8	9	5
5	9	7	4	8	3	1	2	6
1	6	8	2	5	9	4	7	3
4	5	2	8	3	6	7	1	9
7	1	6	9	4	5	3	8	2
8	3	9	1	7	2	5	6	4

Pi Day Sudoku

The basic structure of the classical sudoku problem can easily be extended to formulate more complex puzzles. One such example is the Pi Day sudoku puzzle.

Pi Day is a celebration of the number π that occurs every March 14. In honor of Pi Day, Brainfreeze Puzzles (Riley and Taalman 2008) celebrates this day with a special 12×12 grid sudoku puzzle. The 2008 Pi Day sudoku puzzle is shown in Figure 6.1.3.

Output 6.1.3 Pi Day Sudoku 2008

3			1	5	4			1		9	5
	1			3					1	3	6
		4			3		8			2	
5			1			9	2	5			1
	9			5			5				
5	8	1			9			3		6	
	5		8			2			5	5	3
				5			6			1	
2			5	1	5			5			9
	6			4		1			3		
1	5	1					5			5	
5	5		4			3	1	6			8

The rules for this puzzle are a little different from the rules for standard sudoku:

1. Rather than using regular 3×3 blocks, this puzzle uses jigsaw regions such that highlighted regions in the middle resemble the Greek letter π. Each jigsaw region consists of 12 contiguous cells.
2. The first 12 digits of π are used instead of the digits 1–9. Each row, column, and jigsaw region contains the first 12 digits of π (314159265358) in some order. In other words, there are no 7s; one each of 2, 4, 6, 8, and 9; two each of 1 and 3; and three 5s.

To generalize the original sudoku model:

1. Replace the expression that calculates the starting and ending cells of a region by an array that maps each cell to one region.
2. Replace the all-different constraints with GCC constraints. GCC constraints describe how often each value can be assigned to a set of variables. Conceptually, an all-different constraint is a specialized GCC constraint in which both the lower bound and the upper bound of every value is 1.

The data set Raw contains the partially filled values for the grid. It contains missing values where the cell does not yet contain a number.

```
data Raw;
   input C1-C12;
   datalines;
3  .  .  1  5  4  .  .  1  .  9  5
.  1  .  .  3  .  .  .  .  1  3  6
.  .  4  .  .  3  .  8  .  .  2  .
5  .  .  1  .  .  9  2  5  .  .  1
.  9  .  .  5  .  .  5  .  .  .  .
5  8  1  .  .  9  .  .  3  .  6  .
.  5  .  8  .  .  2  .  .  5  5  3
.  .  .  .  5  .  .  6  .  .  1  .
2  .  .  5  1  5  .  .  5  .  .  9
.  6  .  .  4  .  1  .  .  3  .  .
1  5  1  .  .  .  .  5  .  .  5  .
5  5  .  4  .  .  3  1  6  .  .  8
;
```

The following statements define the GCC constraints in order to find all solutions of the Pi Day sudoku 2008 puzzle:

```
proc optmodel;
   set ROWS = 1..12;
   /* These declarations are inexpensive and improve clarity: */
   set COLS = ROWS, REGIONS = ROWS, CELLS = ROWS cross COLS;

   /* specify a 12x12 array of region identifiers.
      The spacing is just to make the regions easier to visualize. */
   num region{CELLS} = [
       1  1  1    2  2  2  2  2  2    3  3  3
       1  1  1    2  2  2  2  2  2    3  3  3
       1  1   4  4  4  4    5  5  5  5   3  3
       1  1   4  4  4  4    5  5  5  5   3  3
       1  1   4  4  4  4    5  5  5  5   3  3
       6  6  6   7  7   8  8   9  9  10 10 10
       6  6  6   7  7   8  8   9  9  10 10 10
       6  6  6   7  7   8  8   9  9  10 10 10
       6  6  6   7  7   8  8   9  9  10 10 10
      11 11 11   7  7   8  8   9  9  12 12 12
      11 11 11   7  7   8  8   9  9  12 12 12
      11 11 11 11 11 11     12 12 12 12 12 12 ];

   /* Each area must contain two 1's, two 3's, three 5's, no 7's,
      and one for each of other values from 1 to 9. */
                  /* 1 2 3 4 5 6 7 8 9 */
   num nTimes{1..9} = [2 1 2 1 3 1 0 1 1];
   /* For convenience, create a triplet set version of nTimes.
      In this model, GCC's lower and upper bounds are the same. */
   set N_TIMES = setof{ni in 1..9} <ni,nTimes[ni],nTimes[ni]>;

   /* The number assigned to the ith row and jth column. */
   var X {CELLS} >= 1 <= 9 integer;

   /* X[i,j] = c[i,j] if c[i,j] is not missing */
   num c {CELLS};
   read data raw into [_N_] {j in COLS} <c[_N_,j]=col('C'||j)>;
   for {<i,j> in CELLS: c[i,j] ne .}
      fix X[i,j] = c[i,j];

   con RowCon {i in ROWS}:
      gcc({j in COLS} X[i,j], N_TIMES);

   con ColCon {j in COLS}:
      gcc({i in ROWS} X[i,j], N_TIMES);

   con RegionCon {ri in REGIONS}:
      gcc({<i,j> in CELLS: region[i,j] = ri} X[i,j], N_TIMES);

   solve;
```

```
   /* Replicate typical PROC CLP output from PROC OPTMODEL arrays */
   create data pdsout from
      {<i,j> in ROWS cross COLS}<col('X_'||i||'_'||j)=X[i,j]>;
quit;
```

The only solution of the 2008 Pi Day sudoku puzzle is shown in Output 6.1.4.

Output 6.1.4 Solution to Pi Day Sudoku 2008

Pi Day Sudoku 2008

Obs	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
1	3	2	5	1	5	4	6	3	1	8	9	5
2	4	1	5	2	3	8	5	9	5	1	3	6
3	6	1	4	5	9	3	5	8	3	1	2	5
4	5	3	3	1	8	5	9	2	5	6	4	1
5	8	9	2	6	5	1	1	5	4	3	3	5
6	5	8	1	5	2	9	4	3	3	5	6	1
7	1	5	3	8	1	6	2	4	9	5	5	3
8	9	4	5	3	5	1	5	6	8	2	1	3
9	2	3	6	5	1	5	3	1	5	4	8	9
10	3	6	8	9	4	5	1	5	1	3	5	2
11	1	5	1	3	6	3	8	5	2	9	5	4
12	5	5	9	4	3	2	3	1	6	5	1	8

The corresponding completed grid is shown in Figure 6.1.5.

Output 6.1.5 Solution to Pi Day Sudoku 2008

3	2	5	1	5	4	6	3	1	8	9	5
4	1	5	2	3	8	5	9	5	1	3	6
6	1	4	5	9	3	5	8	3	1	2	5
5	3	3	1	8	5	9	2	5	6	4	1
8	9	2	6	5	1	1	5	4	3	3	5
5	8	1	5	2	9	4	3	3	5	6	1
1	5	3	8	1	6	2	4	9	5	5	3
9	4	5	3	5	1	5	6	8	2	1	3
2	3	6	5	1	5	3	1	5	4	8	9
3	6	8	9	4	5	1	5	1	3	5	2
1	5	1	3	6	3	8	5	2	9	5	4
5	5	9	4	3	2	3	1	6	5	1	8

Magic Square

A magic square is an arrangement of the distinct positive integers from 1 to n^2 in an $n \times n$ matrix such that the sum of the numbers of any row, any column, or any main diagonal is the same number, known as the magic constant. The magic constant of a normal magic square depends only on n and has the value $n(n^2+1)/2$.

This example illustrates the use of the MINRMAXC selection strategy, which is controlled by the VARSELECT= option.

```
%macro magic(n);
   proc optmodel;
      num n = &n;
      /* magic constant */
      num sum = n*(n^2+1)/2;
      set ROWS = 1..n;
      set COLS = 1..n;

      /* X[i,j] = entry (i,j) */
      var X {ROWS, COLS} >= 1 <= n^2 integer;

      /* row sums */
      con RowCon {i in ROWS}:
         sum {j in COLS} X[i,j] = sum;

      /* column sums */
      con ColCon {j in COLS}:
         sum {i in ROWS} X[i,j] = sum;

      /* diagonal: upper left to lower right */
      con DiagCon:
         sum {i in ROWS} X[i,i] = sum;

      /* diagonal: upper right to lower left */
      con AntidiagCon:
         sum {i in ROWS} X[n+1-i,i] = sum;

      /* symmetry-breaking */
      con BreakRowSymmetry:
         X[1,1] + 1 <= X[n,1];
      con BreakDiagSymmetry:
         X[1,1] + 1 <= X[n,n];
      con BreakAntidiagSymmetry:
         X[1,n] + 1 <= X[n,1];

      con alldiff(X);

      solve with CLP / varselect=minrmaxc maxtime=3;
   quit;
%mend magic;

%magic(7)
```

The solution is displayed in Output 6.1.6.

Output 6.1.6 Solution of the Magic Square

1	39	24	40	31	38	2
43	4	34	41	42	5	6
18	20	23	29	30	22	33
36	37	25	3	7	35	32
14	19	44	13	47	12	26
17	45	9	21	8	48	27
46	11	16	28	10	15	49

Example 6.2: Alphabet Blocks Problem

This example illustrates the use of the global cardinality constraint (GCC). The alphabet blocks problem consists of finding an arrangement of letters on four alphabet blocks. Each alphabet block has a single letter on each of its six sides. Collectively, the four blocks contain every letter of the alphabet except Q and Z. By arranging the blocks in various ways, the following words should be spelled out: BAKE, ONYX, ECHO, OVAL, GIRD, SMUG, JUMP, TORN, LUCK, VINY, LUSH, and WRAP.

You can formulate this problem as a CSP by representing each of the 24 letters as an integer variable. The domain of each variable is the set $\{1, 2, 3, 4\}$, which represents block1 through block4. The assignment A = 1 indicates that the letter A is on a side of block1. Because each block has six sides, each value v in $\{1, 2, 3, 4\}$ must be assigned to exactly six variables so that each side of a block has a letter on it. This restriction can be formulated as a global cardinality constraint over all 24 variables, with common lower and upper bounds set equal to 6.

Moreover, in order to spell all the words listed previously, the four letters in each of the 12 words must be on different blocks. Another GCC statement that specifies 12 global cardinality constraints enforces these conditions. You can also formulate these restrictions by using 12 all-different constraints. Finally, four FIX statements break the symmetries that blocks are interchangeable. These constraints preset the blocks that contain the letters B, A, K, and E as block1, block2, block3, and block4, respectively.

The complete representation of the problem is as follows:

```
proc optmodel;
   /* Each letter except Q and Z is represented with a variable. */
   /* The domain of each variable is the set of 4 blocks,        */
   /*   or {1, 2, 3, 4} for short.                               */
   set LETTERS = / A B C D E F G H I J K L M N O P R S T U V W X Y /;
   var Block {LETTERS} integer >= 1 <= 4;
   set BLOCKS = 1..4;

   /* There are exactly 6 letters on each alphabet block */
   con SixLettersPerBlock:
      gcc(Block, setof {b in BLOCKS} <b,6,6>);

   /* The letters in each word must be on different blocks. */
   set WORDS = / BAKE ONYX ECHO OVAL GIRD SMUG JUMP TORN LUCK VINY LUSH WRAP /;
   con CanSpell {w in WORDS}:
      gcc({k in 1..length(w)} Block[char(w,k)], setof {b in BLOCKS} <b,0,1>);

   /* Note 2: These restrictions can also be enforced by ALLDIFF constraints:
   con CanSpellv2 {w in WORDS}:
      alldiff({k in 1..length(w)} Block[char(w,k)]);
   */

   /* Breaking the symmetry that blocks can be interchanged by setting
      the block that contains the letter B as block1, the block that
      contains the letter A as block2, etc. */
   for {k in 1..length('BAKE')} fix Block[char('BAKE',k)] = k;

   solve;
   print Block;
quit;
```

The solution to this problem is shown in Output 6.2.1.

Output 6.2.1 Solution to Alphabet Blocks Problem

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	(0)
Objective Type	Constant
Number of Variables	24
Bounded Above	0
Bounded Below	0
Bounded Below and Above	20
Free	0
Fixed	4
Binary	1
Integer	23
Number of Constraints	13
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	0
Linear Range	0
Alldiff	0
Element	0
GCC	13
Lexico (<=)	0
Lexico (<)	0
Pack	0
Reify	0
Constraint Coefficients	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	CLP
Variable Selection	MINR
Objective Function	(0)
Solution Status	Solution Limit Reached
Objective Value	0
Solutions Found	1
Presolve Time	0.00
Solution Time	0.00

Output 6.2.1 *continued*

[1]	Block
A	2
B	1
C	2
D	2
E	4
F	1
G	4
H	3
I	1
J	2
K	3
L	4
M	3
N	2
O	1
P	4
R	3
S	2
T	4
U	1
V	3
W	1
X	3
Y	4

Example 6.3: Work-Shift Scheduling Problem

This example illustrates the use of the GCC constraint to find a feasible solution to a work-shift scheduling problem and then the use of the element constraint to incorporate cost information in order to find a minimum-cost schedule.

Six workers (Alan, Bob, Juanita, Mike, Ravi, and Aisha) are to be assigned to three working shifts. The first shift needs at least one and at most four people; the second shift needs at least two and at most three people; and the third shift needs exactly two people. Alan cannot work on the first shift; Bob can work only on the third shift. The others can work on any shift. The objective is to find a feasible assignment for this problem.

You can model the minimum and maximum shift requirements by using a GCC constraint and formulate the problem as a standard CSP. The variables W[1],. . . ,W[6] identify the shift to which each of the six workers is assigned: Alan, Bob, Juanita, Mike, Ravi, and Aisha.

```
proc optmodel;
   /*  Six workers (Alan, Bob, Juanita, Mike, Ravi and Aisha)
       are to be assigned to 3 working shifts.            */
   set WORKERS = 1..6;
   var W {WORKERS} integer >= 1 <= 3;

   /* The first shift needs at least 1 and at most 4 people;
```

```
      the second shift needs at least 2 and at most 3 people;
      and the third shift needs exactly 2 people. */
   con ShiftNeeds:
      gcc(W, /<1,1,4>,<2,2,3>,<3,2,2>/);

   /* Alan doesn't work on the first shift. */
   con Alan:
      W[1] ne 1;

   /* Bob works only on the third shift. */
   fix W[2] = 3;

   solve;
   print W;
quit;
```

The resulting assignment is shown in Output 6.3.1.

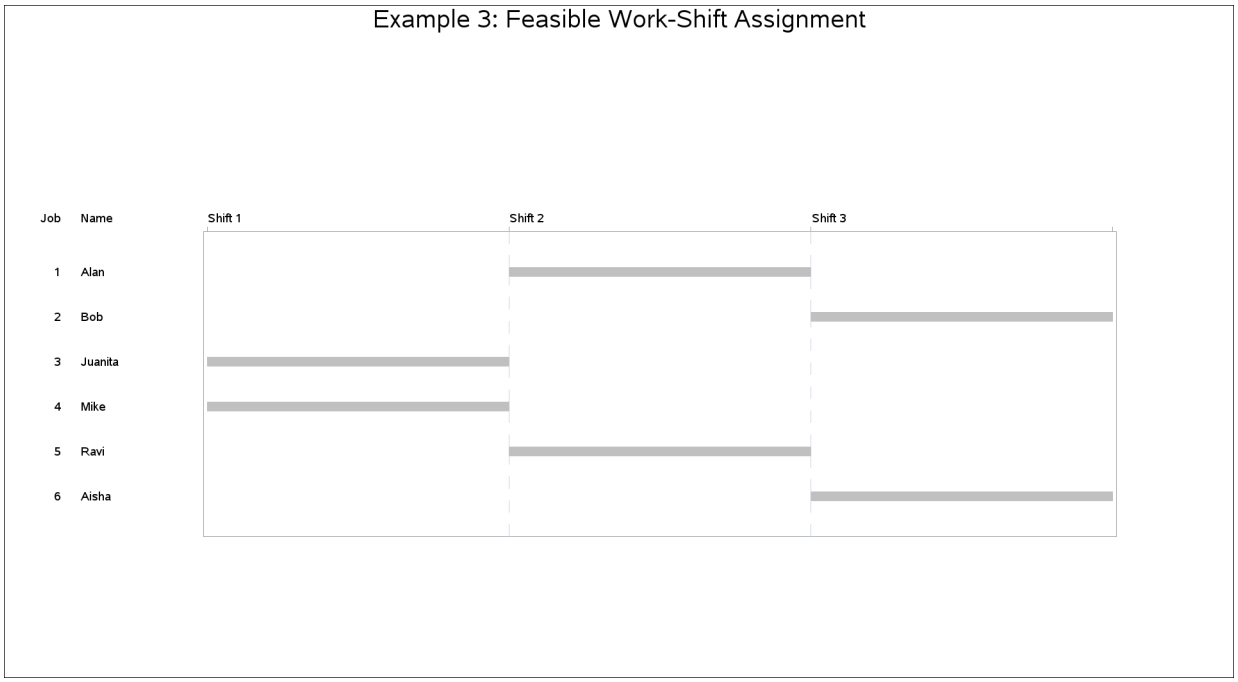
Output 6.3.1 Solution to Work-Shift Scheduling Problem

Solution to Work-Shift Scheduling Problem

Obs	W1	W2	W3	W4	W5	W6
1	2	3	1	1	2	3

A Gantt chart of the corresponding schedule is displayed in Output 6.3.2.

Output 6.3.2 Work-Shift Schedule



Now suppose that every work-shift assignment has a cost associated with it and that the objective of interest is to determine the schedule that has the lowest cost.

The costs of assigning the workers to the different shifts are shown in Table 6.5. A dash ("-") in position (i, j) indicates that worker i cannot work on shift j.

Table 6.5 Costs of Assigning Workers to Shifts

	Shift 1	Shift 2	Shift 3
Alan	-	12	10
Bob	-	-	6
Juanita	16	8	12
Mike	10	6	8
Ravi	6	6	8
Aisha	12	4	4

Based on the cost structure in Table 6.5, the previously derived schedule has a cost of \$54. The objective now is to determine the optimal schedule—one that results in the minimum cost.

Let the variable C_i represent the cost of assigning worker i to a shift. This variable is shift-dependent and is given a high value (for example, 100) if the worker cannot be assigned to a shift. The costs can also be interpreted as preferences if desired. You can use an element constraint to associate the cost C_i with the shift assignment for each worker. For example, C_1, the cost of assigning Alan to a shift, can be determined by the constraint ELEMENT$(W_1, (100, 12, 10), C_1)$.

By adding a linear constraint, $\sum_{i=1}^{n} C_i \leq \text{obj}$, you can limit the solutions to feasible schedules that cost no more than obj. Although an upper bound of \$100 is used in this example, it would suffice to use an upper bound of \$54, the cost of the feasible schedule that was determined earlier.

```
proc optmodel;
   /* Six workers (Alan, Bob, Juanita, Mike, Ravi and Aisha)
      are to be assigned to 3 working shifts.  */
   set WORKERS = 1..6;
   set SHIFTS  = 1..3;
   var W {WORKERS} integer >= 1 <= 3;
   var C {WORKERS} integer >= 1 <= 100;

   /* The first shift needs at least 1 and at most 4 people;
      the second shift needs at least 2 and at most 3 people;
      and the third shift needs exactly 2 people. */
   con GccCon:
      gcc(W, /<1,1,4>,<2,2,3>,<3,2,2>/);

   /* Alan doesn't work on the first shift. */
   con Alan:
      W[1] ne 1;

   /* Bob works only on the third shift. */
   fix W[2] = 3;

   /* Specify the costs of assigning the workers to the shifts.
      Use 100 (a large number) to indicate an assignment
      that is not possible.*/
   num a {WORKERS, SHIFTS} = [
```

```
      100,  12, 10,
      100, 100,  6,
       16,   8, 12
       10,   6,  8
        6,   6,  8
       12,   4,  4
   ];
   con ElementCon {j in WORKERS}:
      element(W[j], {k in SHIFTS} a[j,k], C[j]);

   /* Minimize total cost. */
   min TotalCost = sum {j in WORKERS} C[j];
   con TotalCost_bounds:
      1 <= TotalCost <= 100;

   solve;
   print W;
   create data clpout from
      {j in WORKERS} <col('W'||j)=W[j]> {j in WORKERS} <col('C'||j)=C[j]>;
quit;
```

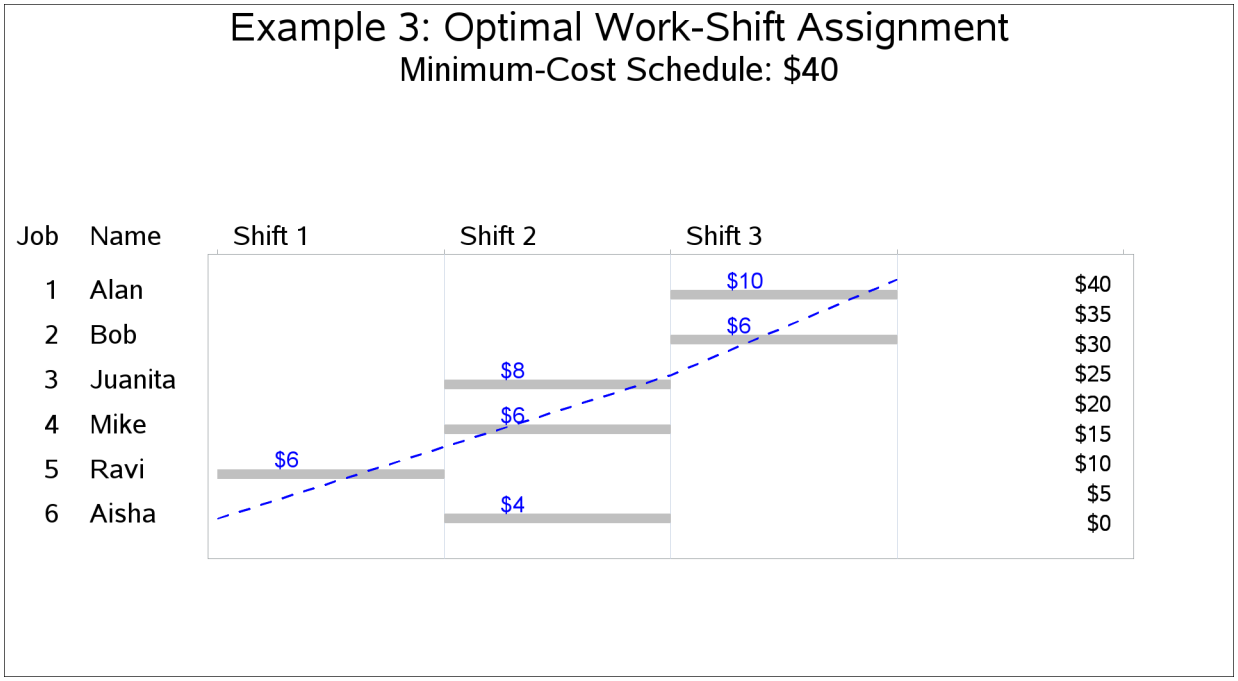
The cost of the optimal schedule, which corresponds to the solution shown in the following output, is $40.

Solution to Optimal Work-Shift Scheduling Problem

Obs	W1	W2	W3	W4	W5	W6	C1	C2	C3	C4	C5	C6
1	3	3	2	2	1	2	10	6	8	6	6	4

The minimum-cost schedule is displayed in the Gantt chart in Output 6.3.3.

Output 6.3.3 Work-Shift Schedule with Minimum Cost



Example 6.4: A Nonlinear Optimization Problem

This example illustrates how you can use the element constraint to represent almost any function between two variables in addition to representing nonstandard domains. Consider the following nonlinear optimization problem:

$$\text{maximize } f(x) = x_1^3 + 5x_2 - 2^{x_3}$$

$$\text{subject to } \begin{cases} x_1 - .5x_2 + x_3^2 \leq 50 \\ \text{mod}(x_1, 4) + .25x_2 \geq 1.5 \end{cases}$$

where x_1 is any integer in [–5, 5], x_2 is any *odd* integer in [–5, 9], and x_3 is any integer in [1, 10].

You can solve this problem by introducing four artificial variables, y_1–y_4, to represent each of the nonlinear terms. Let $y_1 = x_1^3$, $y_2 = 2^{x_3}$, $y_3 = x_3^2$, and $y_4 = \text{mod}(x_1, 4)$. You can represent the domains of x_1 and x_2 (which are not consecutive integers that start from 1) by using element constraints and index variables. For example, any of the following three element constraints specifies that the domain of x_2 is the set of odd integers in $[-5, 9]$:

```
con element(z2,-5 -3 -1 1 3 5 7 9,x2);
con element(z2, {ri in -5..9 by 2} ri, x2);
num range{ri in -5..9 by 2} = ri;
con element(z2,range,x2);
```

Any functional dependencies on x_1 or x_2 can now be defined using z_1 or z_2, respectively, as the index variable in an element constraint. Because the domain of x_3 is $[1, 10]$, you can directly use x_3 as the index variable in an element constraint to define dependencies on x_3.

For example, the following constraint specifies the function $y_1 = x_1^3$, $x_1 \in [-5, 5]$:

```
con element(z1,-125 -64 -27 -8 -1 0 1 8 27 64 125,y1);
/* or, con element(z1, {ri in -5..5} (ri**3), y1); */
num range{ri in -5..5} = ri**3;
con element(z1,range,y1);
```

To solve the problem, define the objective function as demonstrated in the following statements:

```
proc optmodel;
   set DOM{1..3} = [ (-5 .. 5) (-5 .. 9 by 2) (1 .. 10) ];
   var X{i in 1..3} integer >= min{j in DOM[i]} j <= max{j in DOM[i]} j;

   /* map the domain of X[1] and X[2] to 1 .. list size */
   var Z {1..2} integer;
   /* map nonlinear expressions */
   var Y {1..4} integer;

   /* Use an element constraint to represent noncontiguous domains */
   /* domains with negative numbers, and nonlinear functions. */
   /* Z[2] does not appear anywhere else. Its only purpose is
      to restrict X[2] to take a value from DOM[2]. */
   con MapDomainTo1ToCard{i in 1..2}:
      element(Z[i], {k in DOM[i]} k, X[i]);

   /* Functional Dependencies on X[1] */
   /* Y[1] = X[1]^3 -- Use Z[1] for X[1] for proper indexing */
   con Y1:
      element(Z[1], {k in DOM[1]} (k^3), Y[1]);

   /* Y[4] = mod(X[1], 4) */
   con Y4:
      element(Z[1], {k in DOM[1]} (mod(k,4)), Y[4]);


   /* Functional Dependencies on X[3] */
   /* Y[2] = 2^X[3] */
   con Y2:
      element(X[3], {k in DOM[3]} (2^k), Y[2]);

   /* Y[3] = X[3]^2 */
   con Y3:
      element(X[3], {k in DOM[3]} (k^2), Y[3]);

   /* X[1] - 0.5 * X[2] + X[3]^2 <= 50 */
   con Con1:
      X[1] - 0.5 * X[2] + Y[3] <= 50;

   /* mod(X[1],4) + 0.25 * X[2] >= 1.5 */
   con Con2:
      Y[4] + 0.25 * X[2] >= 1.5;

   /* Objective function: X[1]^3 + 5 * X[2] - 2^X[3] */
   max Objective = Y[1] + 5 * X[2] - Y[2];

   solve;
   print X Y Z;
quit;
```

Output 6.4.1 shows the solution that corresponds to the optimal objective value of 168.

Output 6.4.1 Nonlinear Optimization Problem Solution

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Maximization
Objective Function	Objective
Objective Type	Linear
Number of Variables	9
Bounded Above	0
Bounded Below	0
Bounded Below and Above	3
Free	6
Fixed	0
Binary	0
Integer	9
Number of Constraints	8
Linear LE (<=)	1
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Alldiff	0
Element	6
GCC	0
Lexico (<=)	0
Lexico (<)	0
Pack	0
Reify	0
Constraint Coefficients	5

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	CLP
Variable Selection	MINR
Objective Function	Objective
Solution Status	Optimal
Objective Value	168
Solutions Found	1
Presolve Time	0.00
Solution Time	0.00

Output 6.4.1 *continued*

[1]	X	Y	Z
1	5	125	11
2	9	2	8
3	1	1	
4		1	

Example 6.5: Car Painting Problem

The car painting process is an important part of the automobile manufacturing industry. Purging (the act of changing colors during assembly) is expensive because of the added cost of wasted paint and solvents from each color change and the extra time that the purging process requires. The objective of the car painting problem is to sequence the cars in the assembly line in order to minimize the number of paint color changes (Sokol 2002; Trick 2004).

Suppose an assembly line contains 10 cars, which are ordered 1, 2, . . . , 10. A car must be painted within three positions of its assembly order. For example, car 5 can be painted in positions 2 through 8. Cars 1, 5, and 9 are red; 2, 6, and 10 are blue; 3 and 7 green; and 4 and 8 are yellow. The initial sequence 1, 2, . . . , 10 corresponds to the color pattern RBGYRBGYRB and has nine purgings. The objective is to find a solution that minimizes the number of purgings.

This problem can be formulated as a CSP as follows:

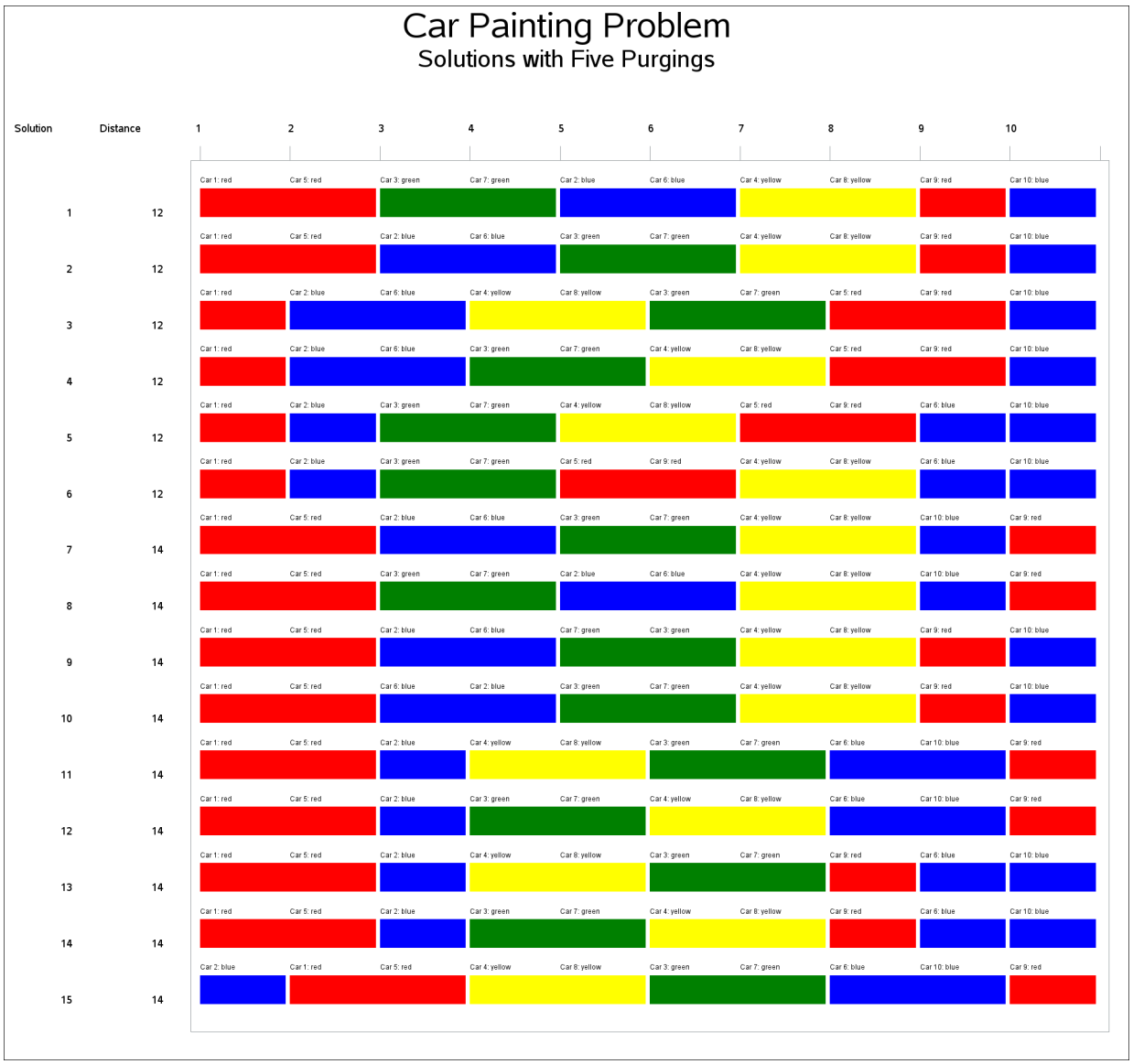
- The input is the color of each car currently on the assembly line.
- The output variables are the slot in which each car will be painted and whether purging will be required after that painting operation.
- Set the bounds of the slot variables to their feasible range, at most three slots before or after the car's current position.
- To determine whether purging is needed, use another variable to store the color of the car painted in each slot.
- Use an element constraint to determine the color used in each slot from the car assigned to that slot.
- Use a reify constraint to determine whether two consecutive slots are assigned different colors, and thus whether purging is required.
- Finally, use a linear constraint to limit the total number of purgings.

The following %CAR_PAINTING macro determines all feasible solutions for the number of purgings that is specified as a parameter to the macro:

```
%macro car_painting(purgings);
   proc optmodel;
      num nCars = 10;
      /* a car is identified by its original slots */
      set SLOTS = 1..nCars;

      /* maximum reshuffling of any car on the line*/
      num maxMove init 3;
      /* which car is in slot i. */
      var S {si in SLOTS} integer >= max(1, si - maxMove)
                                  <= min(nCars, si + maxMove) ;

      /* which color the car in slot i is. */
      /* Red=1; Blue=2; Green=3; Yellow=4 */
      num nColors=4;
      num colorOf{SLOTS} = [ 1 2 3 4 1 2 3 4 1 2 ];
      var C {SLOTS} integer >= 1 <= nColors;

      con ElementCon {i in SLOTS}:
         element(S[i], colorOf, C[i]);

      /* A car can be painted only once. */
      con PaintOnlyOnce:
         alldiff(S);

      /* Whether there is a purge after slot i.
        You can ignore any purging that would happen at the end of the shift. */
      var P {SLOTS diff {nCars}} binary;

      con ReifyCon {i in SLOTS diff {nCars}}:
         reify(P[i], C[i] ne C[i+1]);

      /* Calculate the number of purgings. */
      con PurgingsCon:
         sum {i in SLOTS diff {nCars}} P[i] <= &purgings;

      solve with CLP / findall;
      /* Replicate typical PROC CLP output from PROC OPTMODEL arrays */
      create data car_ds(drop=k) from [k]=(1.._NSOL_)
         {i in SLOTS} <col('S'||i)=S[i].sol[k]>
         {i in SLOTS} <col('C'||i)=C[i].sol[k]>;
   quit;
%mend;
%car_painting(5);
```

The problem is infeasible for four purgings. The CLP solver finds 87 possible solutions for the five-purgings problem. The solutions are sorted by the total distance that all cars are moved in the sequencing, which ranges from 12 to 22 slots. The first 15 solutions are displayed in the Gantt chart in Output 6.5.1. Each row represents a solution, and each color transition represents a paint purging.

Output 6.5.1 Car Painting Schedule with Five Purgings



Example 6.6: Scene Allocation Problem

The scene allocation problem consists of deciding when to shoot each scene of a movie in order to minimize the total production cost (Van Hentenryck 2002). Each scene involves a number of actors, and at most five scenes a day can be shot. All actors who appear in a scene must be present in the studio on the day the scene is shot. Each actor earns a daily rate for each day spent in the studio, regardless of the number of scenes in which he or she appears on that day. The goal is to shoot the movie for the lowest possible production cost.

The actors' names, their daily fees, and the scenes in which they appear are contained in the Scene data set, which is shown in Output 6.6.1. The data set variables S_Var1, . . . , S_Var9 indicate the scenes in which the

actor appears. For example, the first observation indicates that Patt's daily fee is 26,481 and that Patt appears in scenes 2, 5, 7, 10, 11, 13, 15, and 17.

Output 6.6.1 Scene Data Set

Obs	Number	Actor	DailyFee	S_Var1	S_Var2	S_Var3	S_Var4	S_Var5	S_Var6	S_Var7	S_Var8	S_Var9
1	1	Patt	26481	2	5	7	10	11	13	15	17	.
2	2	Casta	25043	4	7	9	10	13	16	19	.	.
3	3	Scolaro	30310	3	6	9	10	14	16	17	18	.
4	4	Murphy	4085	2	8	12	13	15	.	.	.	.
5	5	Brown	7562	2	3	12	17	.	.	.	.	.
6	6	Hacket	9381	1	2	12	13	18	.	.	.	.
7	7	Anderson	8770	5	6	14	.	.	.	.	.	.
8	8	McDougal	5788	3	5	6	9	10	12	15	16	18
9	9	Mercer	7423	3	4	5	8	9	16	.	.	.
10	10	Spring	3303	5	6	.	.	.	.	.	.	.
11	11	Thompson	9593	6	9	12	15	18	.	.	.	.

There are 19 scenes. At most 5 scenes can be filmed in one day, so at least four days are needed to schedule all the scenes ($\lceil \frac{19}{5} \rceil = 4$). Let S_{jk} be a binary variable that equals 1 if scene j is shot on day k. Let A_{ik} be another binary variable that equals 1 if actor i is present on day k. The input daily_fee$_i$ is the daily cost of the ith actor.

The objective function that represents the total production cost is

$$\min \sum_{i=1}^{11} \sum_{k=1}^{4} \text{daily\_fee}_i \times A_{ik}$$

This example illustrates the use of symmetry-breaking constraints. In this model, the "1" in day 1 does not refer to sequence but simply to the label of the day. Thus, you can call day 1 the day on which scene 1 is shot, whichever day that is. Similarly, either scene 2 is shot on the same day as scene 1 (day 1) or it is shot on another day, which you can call day 2. Scene 3 is shot either on one of those two days or on another day. Adding constraints that eliminate symmetry can significantly improve the performance of a CLP model. In this model, the symmetry-breaking constraints prevent the solver from considering three other assignments that do not differ in any meaningful way.

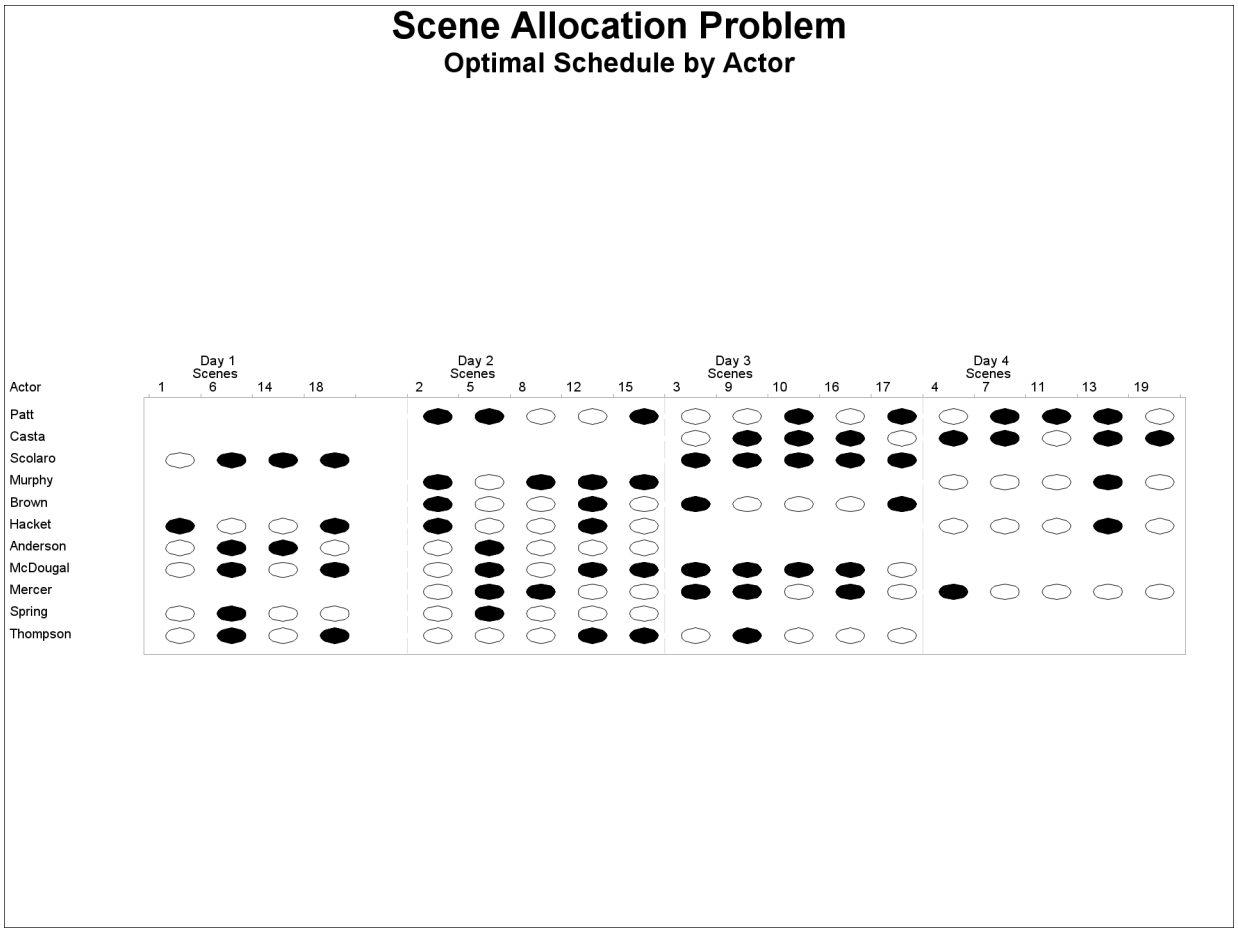
The following PROC OPTMODEL statements implement these ideas:

```
proc optmodel;
   set ACTORS;
   str actor_name {ACTORS};
   num daily_fee {ACTORS};
   num most_scenes = 9; /* most scenes by any actor */
   num scene_list {ACTORS, 1..most_scenes};
   read data scene into ACTORS=[_N_]
      actor_name=Actor daily_fee=DailyFee
      {j in 1..most_scenes} <scene_list[_N_,j]=col('S_Var'||j)>;
   print actor_name daily_fee scene_list;

   set SCENES_actor {i in ACTORS} =
      (setof {j in 1..most_scenes} scene_list[i,j]) diff {.};
```

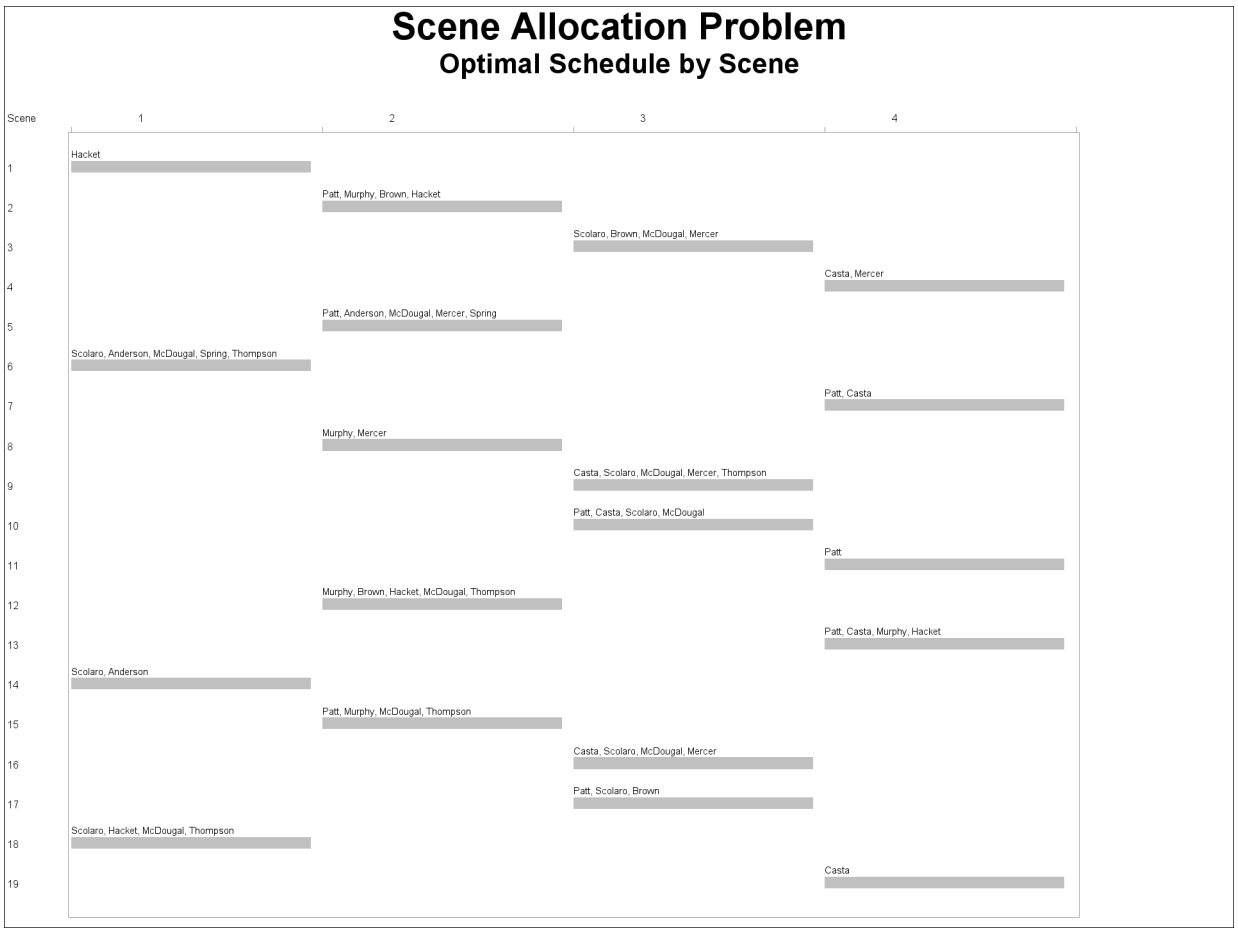
```
   set SCENES = 1..19;
   set DAYS = 1..4;

   /* Indicates if actor i is present on day k. */
   var A {ACTORS, DAYS} binary;

   /* Indicates if scene j is shot on day k. */
   var S {SCENES, DAYS} binary;

   /* Every scene is shot exactly once.*/
   con SceneCon {j in SCENES}:
      gcc({k in DAYS} S[j,k], {<1,1,1>});

   /* At least 4 and at most 5 scenes are shot per day. */
   con NumScenesPerDayCon {k in DAYS}:
      gcc({j in SCENES} S[j,k], {<1,4,5>});

   /* Actors for a scene must be present on day of shooting. */
   con LinkCon {i in ACTORS, j in SCENES_actor[i], k in DAYS}:
      S[j,k] <= A[i,k];

   /* symmetry-breaking constraints. Without loss of any generality, we
      can assume Scene1 to be shot on day 1, Scene2 to be shot on day 1
      or day 2, and Scene3 to be shot on either day 1, day 2, or day 3.  */
   fix S[1,1] = 1;
   for {k in 2..4} fix S[1,k] = 0;
   for {k in 3..4} fix S[2,k] = 0;
   fix S[3,4] = 0;

   /* If Scene2 is shot on day 1, (as opposed to day 2) */
   /* then Scene3 can be shot on day 1 or day 2 (but not day 3). */
   con Symmetry:
      S[2,1] + S[3,3] <= 1;

   /* Minimize total cost. */
   min TotalCost = sum {i in ACTORS, k in DAYS} daily_fee[i] * A[i,k];

   /* Set lower and upper bounds for the objective value   */
   /* Lower bound: every actor appears on one day.         */
   /* Upper bound: every actor appears on all four days.   */
   num obj_lb = sum {i in ACTORS} daily_fee[i];
   num obj_ub = sum {i in ACTORS, k in DAYS} daily_fee[i];
   put obj_lb= obj_ub=;
   con TotalCost_bounds:
      obj_lb <= TotalCost <= obj_ub;

   solve with CLP / varselect=maxc;
quit;
```

The optimal production cost is 334,144, as reported in the _OROPTMODEL_ macro variable. The corresponding actor schedules and scene schedules are displayed in Output 6.6.2 and Output 6.6.3, respectively.

Output 6.6.2 Scene Allocation Problem: Actor Schedules



Output 6.6.3 Scene Allocation Problem: Scene Schedules



Example 6.7: Car Sequencing Problem

This problem is an instance of a category of problems known as the car sequencing problem. A considerable amount of literature discusses this problem (Dincbas, Simonis, and Van Hentenryck 1988; Gravel, Gagne, and Price 2005; Solnon et al. 2008).

A number of cars are to be produced on an assembly line. Each car is customized to include a specific set of options, such as air conditioning, a sunroof, a navigation system, and so on. The assembly line moves through several workstations for installation of these options. The cars cannot be positioned randomly, because each workstation has limited capacity and needs time to set itself up to install the options as the car moves in front of the station. These capacity constraints are formalized using constraints of the form *m* out of *N*, which indicates that the workstation can install the option on *m* out of every sequence of *N* cars. The car sequencing problem is to determine a sequencing of the cars on the assembly line that satisfies the demand constraints for each set of car options and the capacity constraints for each workstation.

This example comes from Dincbas, Simonis, and Van Hentenryck (1988). Ten cars need to be customized with five possible options. A class of car is defined by a specific set of options; there are six classes of cars.

The data are presented in Table 6.6.

Table 6.6 Option Installation Data

Option Name	Option Type	Capacity m/N	Car Class 1	2	3	4	5	6
Option 1	1	1/2	1	0	0	0	1	1
Option 2	2	2/3	0	0	1	1	0	1
Option 3	3	1/3	1	0	0	0	1	0
Option 4	4	2/5	1	1	0	1	0	0
Option 5	5	1/5	0	0	1	0	0	0
Number of Cars			**1**	**1**	**2**	**2**	**2**	**2**

For example, car class 4 requires installation of option 2 and option 4, and two cars of this class are required. The workstation for option 2 can process only two out of every sequence of three cars. The workstation for option 4 has even less capacity—two out of every five cars.

The data for this problem are used to create a SAS data set, which drives the generation of variables and constraints in PROC OPTMODEL.

The decision variables for this problem are shown in Table 6.7.

Table 6.7 Decision Variables

Variable Definition	Description
S{SLOTS} >=1 <= 6	S_i is the class of cars assigned to slot i.
var O {SLOTS, OPTIONS} binary	$O_{ij} = 1$ if the class assigned to slot i needs option j.

The following SAS statements express the workstation capacity constraints by using a set of linear constraints for each workstation. A single GCC constraint expresses the demand constraints for each car class. An element constraint for each option variable expresses the relationships between slot variables and option variables.

This model also includes a set of redundant constraints, in the sense that the preceding logical constraints correctly represent the set of feasible solutions. However, the redundant constraints provide the solver with further information specific to this problem, significantly improving the efficiency of domain propagation. Redundant constraints are a core fixture of CLP models. They can determine whether a model will linger and be unsolvable for real data or will produce instant results.

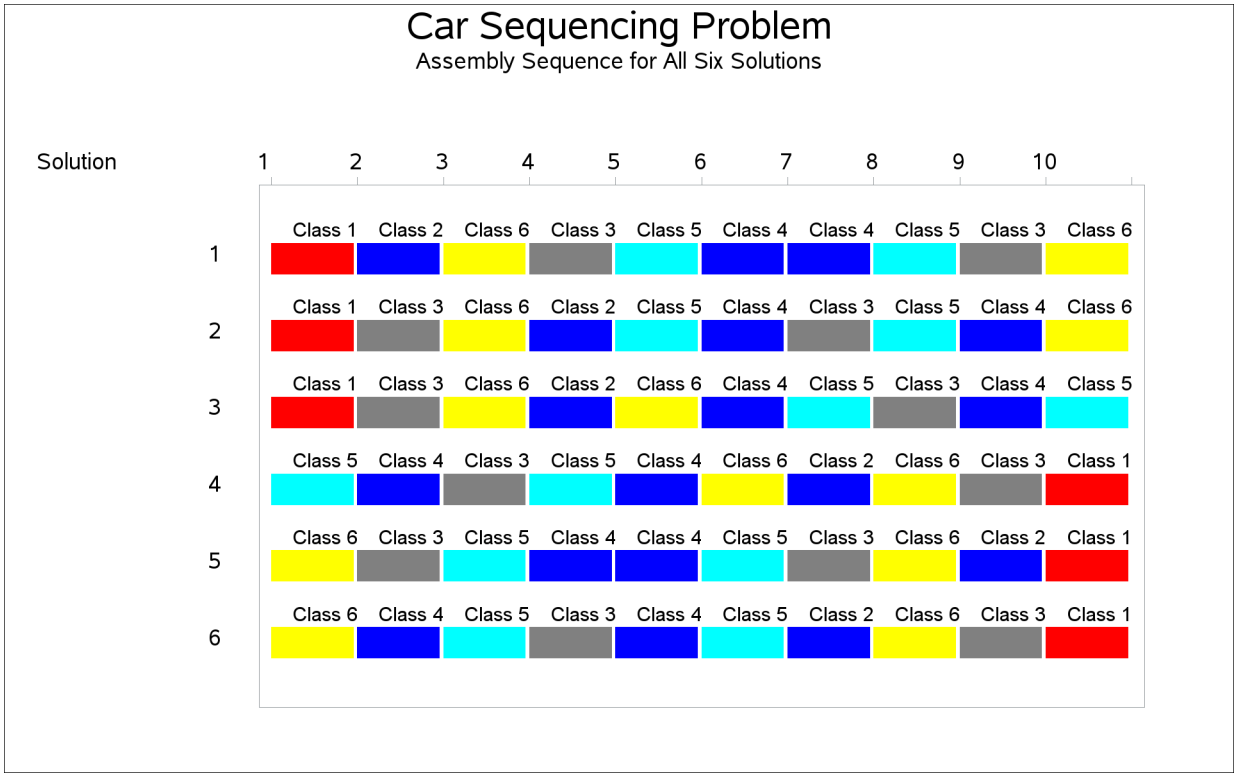
The idea behind the redundant constraint in this model is the following realization: if the workstation for option j has capacity r out of s, then at most r cars in the sequence $(n-s+1),\ldots,n$ can have option j, where n is the total number of cars. Consequently, at least $n_j - r$ cars in the sequence $1,\ldots,n-s$ must have option j, where n_j is the number of cars that have option j. Generalizing this further, at least $n_j - k \times r$ cars in the sequence $1,\ldots,(n-k\times s)$ must have option j, $k = 1,\ldots,\lfloor n/s \rfloor$.

```
data class_data;
   input class cars_cls;
   datalines;
1 1
2 1
3 2
4 2
5 2
6 2
;

data option_data;
   input option max blSz class1-class6;
   datalines;
1 1 2 1 0 0 0 1 1
2 2 3 0 0 1 1 0 1
3 1 3 1 0 0 0 1 0
4 2 5 1 1 0 1 0 0
5 1 5 0 0 1 0 0 0
;

%macro car_sequencing(outdata);
   proc optmodel;
      set CLASSES;
      num nClasses = card(CLASSES);
      num cars_cls {CLASSES};
      read data class_data into CLASSES=[class] cars_cls;

      set OPTIONS;
      num max {OPTIONS};
      num blSz {OPTIONS};
      num list {OPTIONS, CLASSES};
      num cars_opt {i in OPTIONS} = sum {k in CLASSES} cars_cls[k] * list[i,k];
      read data option_data into OPTIONS=[option] max blSz
         {k in CLASSES} <list[option,k]=col('class'||k)>;

      num nCars = sum {k in CLASSES} cars_cls[k];
      set SLOTS = 1..nCars;

      /* Declare Variables */
      /* Slot variables: S[i] - class of car assigned to Slot i */
      var S {SLOTS} integer >= 1 <= nClasses;

      /* Option variables: O[i,j]
      - indicates if class assigned to Slot i needs Option j */
      var O {SLOTS, OPTIONS} binary;

      /* Capacity Constraints: for each option j */
      /* Install in at most max[j] out of every sequence of blSz[j] cars */
      con CapacityCon {j in OPTIONS, i in 0..(nCars-blSz[j])}:
         sum {k in 1..blSz[j]} O[i+k,j] <= max[j];

      /* Demand Constraints: for each class k */
```

```
      /* Exactly cars_cls[k] cars */
      con MeetDemandCon:
         gcc(S, setof{k in CLASSES} <k,cars_cls[k],cars_cls[k]>);

      /* Element Constraints: For each slot i and each option j */
      /* relate the slot variable to the option variables.      */
      /* O[i,j] = list[j,S[i]]                                  */
      con OptionsAtSlotCon {i in SLOTS, j in OPTIONS}:
         element(S[i], {k in CLASSES} list[j,k], O[i,j]);

      /* Redundant Constraints to improve efficiency - for every  */
      /*    option j.                                             */
      /* At most max[j] out of every sequence of blSz[j] cars     */
      /*    requires option j.                                    */
      /* All the other slots contain at least cars_opt[j] - max[j]*/
      /*    cars with option j                                    */
      con BoundRemainingCon {j in OPTIONS, i in 1..(nCars/blSz[j])}:
         sum {k in 1..(nCars-i*blSz[j])} O[k,j] >= cars_opt[j] - i * max[j];

      solve with CLP / varselect=minrmaxc findall;

      /* Replicate typical PROC CLP output from PROC OPTMODEL arrays */
      create data &outdata.(drop=sol) from [sol]=(1.._NSOL_)
         {i in SLOTS} <col('S_'||i)=S[i].sol[sol]>
         {i in SLOTS, j in OPTIONS} <col('O_'||i||'_'||j)=O[i,j].sol[sol]>;
   quit;
%mend;
%car_sequencing(sequence_out);
```

This problem has six solutions, as shown in Output 6.7.1.

Output 6.7.1 Car Sequencing



Example 6.8: Balanced Incomplete Block Design

Balanced incomplete block design (BIBD) generation is a standard combinatorial problem from design theory. The concept was originally developed in the design of statistical experiments; applications have expanded to other fields, such as coding theory, network reliability, and cryptography. A BIBD is an arrangement of v distinct objects into b blocks such that the following conditions are met:

- Each block contains exactly k distinct objects.
- Each object occurs in exactly r different blocks.
- Every two distinct objects occur together in exactly λ blocks.

A BIBD is therefore specified by its parameters (v, b, r, k, λ). It can be proved that when a BIBD exists, its parameters must satisfy the following conditions:

- $rv = bk$
- $\lambda(v-1) = r(k-1)$
- $b \geq v$

The preceding conditions are not sufficient to guarantee the existence of a BIBD (Prestwich 2001). For example, the parameters $(15, 21, 7, 5, 2)$ satisfy the preceding conditions, but a BIBD that has these parameters does not exist. Computational methods of BIBD generation usually suffer from combinatorial explosion, in part because of the large number of symmetries: for any solution, any two objects or blocks can be exchanged to obtain another solution.

This example demonstrates how to express a BIBD problem as a CSP and how to use lexicographic ordering constraints to break symmetries. Note that this example is for illustration only. SAS provides an autocall macro, MKTBIBD, for solving BIBD problems. The most direct CSP model for BIBD, as described in Meseguer and Torras (2001), represents a BIBD as a $v \times b$ matrix $\boldsymbol{X}$. Each matrix entry is a Boolean decision variable $X_{i,c}$ that satisfies $X_{i,c} = 1$ if and only if block c contains object i. The condition that each object occurs in exactly r blocks (or, equivalently, that there are r 1s per row) can be expressed as v linear constraints:

$$\sum_{c=1}^{b} X_{i,c} = r \quad \text{for } i = 1, \ldots, v$$

Alternatively, you can use global cardinality constraints to ensure that there are exactly $b - r$ 0s and r 1s in $X_{i,1}, \ldots, X_{i,b}$ for each object i:

$$\text{gcc}(X_{i,1}, \ldots, X_{i,b}) = ((0, 0, b-r)(1, 0, r)) \quad \text{for } i = 1, \ldots, v$$

Similarly, you can use the following constraints to specify the condition that each block contain exactly k objects (there are k 1s per column):

$$\text{gcc}(X_{1,c}, \ldots, X_{v,c}) = ((0, 0, v-k)(1, 0, k)) \quad \text{for } c = 1, \ldots, b$$

To enforce the final condition that every two distinct objects occur together in exactly λ blocks (equivalently, that the scalar product of every pair of rows equal λ), you can introduce the auxiliary variables $P_{i,j,c}$ for

every $i < j$, which indicate whether objects i and j both occur in block c. The following reify constraint ensures that $P_{i,j,c} = 1$ if and only if block c contains both objects i and j:

$$\text{reify } P_{i,j,c} : (X_{i,c} + X_{j,c} = 2)$$

The following constraints ensure that the final condition holds:

$$\text{gcc}(P_{i,j,1}, \ldots, P_{i,j,b}) = ((0, 0, b - \lambda)(1, 0, \lambda)) \quad \text{for } i = 1, \ldots, v - 1 \text{ and } j = i + 1, \ldots, v$$

The objects and the blocks are interchangeable, so the matrix $\boldsymbol{X}$ has total row symmetry and total column symmetry. Because of the constraints on the rows, no pair of rows can be equal unless $r = \lambda$. To break the row symmetry, you can impose strict lexicographic ordering on the rows of X as follows:

$$(X_{i,1}, \ldots, X_{i,b}) <_{\text{lex}} (X_{i-1,1}, \ldots, X_{i-1,b}) \quad \text{for } i = 2, \ldots, v$$

To break the column symmetry, you can impose lexicographic ordering on the columns of X as follows:

$$(X_{1,c}, \ldots, X_{v,c}) \leq_{\text{lex}} (X_{1,c-1}, \ldots, X_{v,c-1}) \quad \text{for } c = 2, \ldots, b$$

The following SAS macro incorporates all the preceding constraints. For the specified parameters (v, b, r, k, λ), the macro either finds BIBDs or proves that a BIBD does not exist.

```
%macro bibd(v, b, r, k, lambda, out=bibdout);
   /* Arrange v objects into b blocks such that:
         (i) each object occurs in exactly r blocks,
         (ii) each block contains exactly k objects,
         (iii) every pair of objects occur together in exactly lambda blocks.

      Equivalently, create a binary matrix with v rows and b columns,
      with r 1s per row, k 1s per column,
      and scalar product lambda between any pair of distinct rows.
   */

   /* Check necessary conditions */
   %if (%eval(&r * &v) ne %eval(&b * &k)) or
      (%eval(&lambda * (&v - 1)) ne %eval(&r * (&k - 1))) or
      (&v > &b) %then %do;
      %put BIBD necessary conditions are not met.;
      %goto EXIT;
   %end;

   proc optmodel;
      num v = &v;
      num b = &b;
      num r = &r;
      num k = &k;
      num lambda = &lambda;
      set OBJECTS = 1..v;
      set BLOCKS = 1..b;

      /* Decision variable X[i,c] = 1 iff object i occurs in block c. */
      var X {OBJECTS, BLOCKS} binary;
```

```
      /* Mandatory constraints: */
      /* (i) Each object occurs in exactly r blocks. */
      con Exactly_r_blocks {i in OBJECTS}:
         gcc({c in BLOCKS} X[i,c], {<0,0,b-r>,<1,0,r>});

      /* (ii) Each block contains exactly k objects. */
      con Exactly_k_objects {c in BLOCKS}:
         gcc({i in OBJECTS} X[i,c], {<0,0,v-k>,<1,0,k>});

      /* (iii) Every pair of objects occurs in exactly lambda blocks. */
      set PAIRS = {i in OBJECTS, j in OBJECTS: i < j};
      /* auxiliary variable P[i,j,c] = 1 iff both i and j occur in c */
      var P {PAIRS, BLOCKS} binary;
      con Pairs_reify {<i,j> in PAIRS, c in BLOCKS}:
         reify(P[i,j,c], X[i,c] + X[j,c] = 2);
      con Pairs_gcc {<i,j> in PAIRS}:
         gcc({c in BLOCKS} P[i,j,c], {<0,0,b-lambda>,<1,0,lambda>});

      /* symmetry-breaking constraints: */
      /* Break row symmetry via lexicographic ordering constraints. */
      con Symmetry_i {i in OBJECTS diff {1}}:
         lexico({c in BLOCKS} X[i,c] < {c in BLOCKS} X[i-1,c]);

      /* Break column symmetry via lexicographic ordering constraints. */
      con Symmetry_c {c in BLOCKS diff {1}}:
         lexico({i in OBJECTS} X[i,c] <= {i in OBJECTS} X[i,c-1]);

      solve with CLP / varselect=FIFO;
      create data &out from
         {i in OBJECTS, c in BLOCKS} <col('X'||i||'_'||c)=X[i,c]>;
   quit;
   %put &_oroptmodel_;
%EXIT:
%mend bibd;
```

The following statement invokes the macro to find a BIBD design for the parameters $(15, 15, 7, 7, 3)$:

```
%bibd(15,15,7,7,3);
```

The output is displayed in Output 6.8.1.

Output 6.8.1 Balanced Incomplete Block Design for (15,15,7,7,3)

Balanced Incomplete Block Design Problem
(15, 15, 7, 7, 3)

Obs	Block1	Block2	Block3	Block4	Block5	Block6	Block7	Block8	Block9	Block10	Block11
1	1	1	1	1	1	1	1	0	0	0	0
2	1	1	1	0	0	0	0	1	1	1	1
3	1	1	0	1	0	0	0	1	0	0	0
4	1	0	1	0	1	0	0	0	1	0	0
5	1	0	0	1	0	1	0	0	0	1	1
6	1	0	0	0	1	0	1	0	0	1	1
7	1	0	0	0	0	1	1	1	1	0	0
8	0	1	1	0	0	0	1	0	0	1	0
9	0	1	0	1	0	0	1	0	1	0	1
10	0	1	0	0	1	1	0	1	0	1	0
11	0	1	0	0	1	1	0	0	1	0	1
12	0	0	1	1	1	0	0	1	0	0	1
13	0	0	1	1	0	1	0	0	1	1	0
14	0	0	1	0	0	1	1	1	0	0	1
15	0	0	0	1	1	0	1	1	1	1	0

Obs	Block12	Block13	Block14	Block15
1	0	0	0	0
2	0	0	0	0
3	1	1	1	0
4	1	1	0	1
5	1	0	0	1
6	0	1	1	0
7	0	0	1	1
8	1	0	1	1
9	0	1	0	1
10	0	1	0	1
11	1	0	1	0
12	0	0	1	1
13	0	1	1	0
14	1	1	0	0
15	1	0	0	0

Example 6.9: Progressive Party Problem

This example demonstrates the use of the pack constraint to solve an instance of the progressive party problem (Smith et al. 1996). In the original progressive party problem, a number of yacht crews and their boats congregate at a yachting rally. In order for each crew to socialize with as many other crews as possible, some of the boats are selected to serve as "host boats" for six rounds of parties. The crews of the host boats stay with their boats for all six rounds. The crews of the remaining boats, called "guest crews," are assigned to visit a different host boat in each round.

Given the number of boats at the rally, the capacity of each boat, and the size of each crew, the objective of the original problem is to assign all the guest crews to host boats for each of the six rounds, using the minimum number of host boats. The partitioning of crews into guests and hosts is fixed throughout all rounds. No two crews should meet more than once. The assignments are constrained by the spare capacities (total capacity minus crew size) of the host boats and the crew sizes of the guest boats. Some boats cannot be hosts (zero spare capacity), and other boats must be hosts.

In this instance of the problem, the designation of the minimum requirement of 13 hosts is assumed (boats 1 through 12 and 14). The formulation solves up to eight rounds, but only two rounds are scheduled for this example. The total capacities and crew sizes of the boats are shown in Output 6.9.1.

Output 6.9.1 Progressive Party Problem Input

Progressive Party Problem Input

boatnum	capacity	crewsize
1	6	2
2	8	2
3	12	2
4	12	2
5	12	4
6	12	4
7	12	4
8	10	1
9	10	2
10	10	2
11	10	2
12	10	3
13	8	4
14	8	2
15	8	3
16	12	6
17	8	2
18	8	2
19	8	4
20	8	2
21	8	4
22	8	5
23	7	4
24	7	4
25	7	2
26	7	2
27	7	4
28	7	5
29	6	2
30	6	4
31	6	2
32	6	2
33	6	2
34	6	2
35	6	2
36	6	2
37	6	4
38	6	5
39	9	7
40	0	2
41	0	3
42	0	4

The following statements and DATA steps process the data and designate host boats:

```
data hostability;
   set capacities;
   spareCapacity = capacity - crewsize;
run;

data hosts guests;
   set hostability;
   if (boatnum <= 12 or boatnum eq 14) then do;
      output hosts;
   end;
   else do;
      output guests;
   end;
run;

/* sort so guest boats with larger crews appear first */
proc sort data=guests;
   by descending crewsize;
run;

data capacities;
   format boatnum capacity 2.;
   set hosts guests;
   seqno = _n_;
run;
```

To model the progressive party problem for the CLP solver, first define the following sets of variables:

- Item variables x_{it} contain the host boat number for the assignment of guest boat i in round t.
- Load variables L_{ht} contain the load of host boat h in round t.
- Variable m_{ijt} are binary variables that take a value of 1 if and only if guest boats i and j are assigned to the same host boat in round t.

Next, describe the set of constraints that are used in the model:

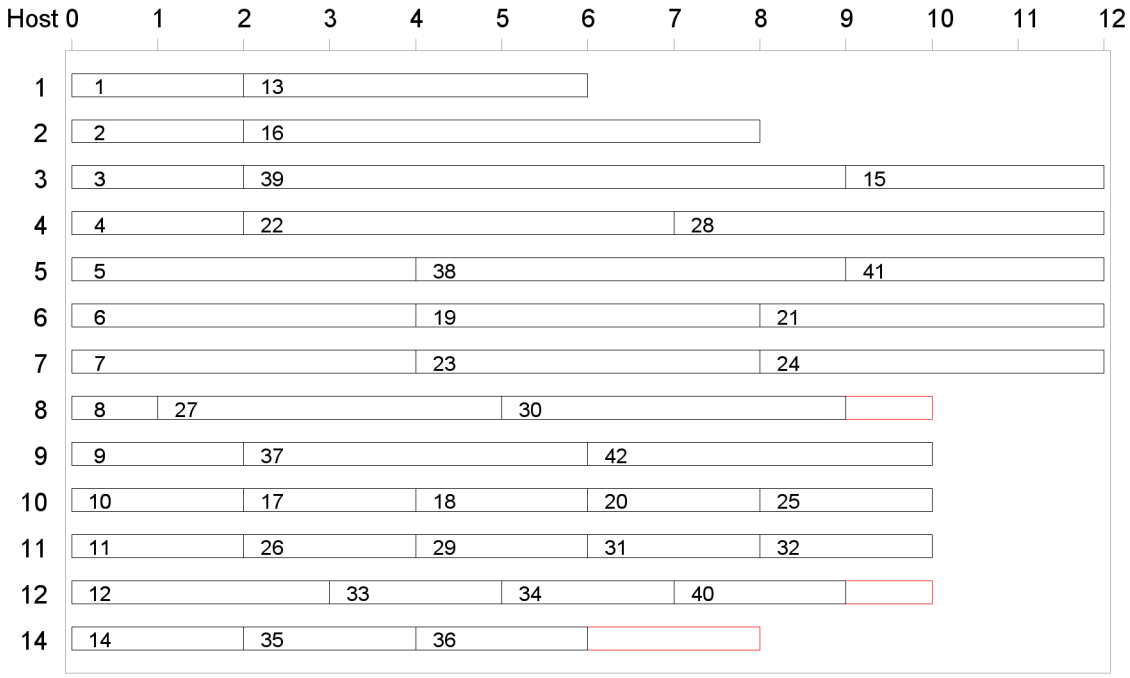
- All-different constraints ensure that a guest boat is not assigned to the same host boat in different rounds.
- Reify constraints regulate the values that are assigned to the aforementioned indicator variables m_{ijt}.
- The reified indicator variables appear in linear constraints to enforce the requirement to meet no more than once.
- One pack constraint per round maintains the capacity limits of the host boats.
- Finally, a symmetry-breaking linear constraint orders the host boat assignments for the highest-numbered guest boat across rounds.

The following statements call PROC OPTMODEL to define the variables, specify the constraints, and solve the problem:

```
%let rounds=2;
%let numhosts=13;
proc optmodel;
   num numrounds = &rounds;
   set ROUNDS = 1..numrounds;
   num numhosts = &numhosts;
   set HOSTS = 1..numhosts;
   set BOATS;
   num numboats = card(BOATS);
   num capacity {BOATS};
   num crewsize {BOATS};
   num spareCapacity{hi in HOSTS} = capacity[hi] - crewsize[hi];
   /* Use the descending crew order for guests (seqno)
      rather than the actual boat id (boatnum)
      to help the performance of the PACK predicate. */
   read data capacities into BOATS=[seqno] capacity crewsize;

   /* Assume that the first numhosts boats are hosts,
      and process each round in turn.
      X is the host assigned to non-host i for round t. */
   var X {numhosts+1..numboats, ROUNDS} integer >= 1 <= numhosts;
   /* The load of the host boat. */
   var L {hi in HOSTS, ROUNDS} integer >= 0 <= spareCapacity[hi];

   /* Assign different hosts each round. */
   con AlldiffCon {i in numhosts+1..numboats}:
      alldiff({t in ROUNDS} X[i,t]);

   /* Two crews cannot meet more than once. */
   var M {i in numhosts+1..numboats-1, j in i+1..numboats, t in ROUNDS} binary;
   con ReifyCon {i in numhosts+1..numboats-1, j in i+1..numboats, t in ROUNDS}:
      reify(M[i,j,t], X[i,t] = X[j,t]);
   con Assign {i in numhosts+1..numboats-1, j in i+1..numboats}:
      sum {t in ROUNDS} M[i,j,t] <= 1;

   /* Honor capacities. */
   con PackCon {t in ROUNDS}:
      pack(
         {i in numhosts+1..numboats} X[i,t],
         {i in numhosts+1..numboats} crewsize[i],
         {h in HOSTS} L[h,t]
      );

   /* Break symmetries. */
   con SymmetryCon {t in 1..numrounds-1}:
      X[numboats,t] < X[numboats,t+1];

   solve with CLP / varselect=FIFO;
quit;
```

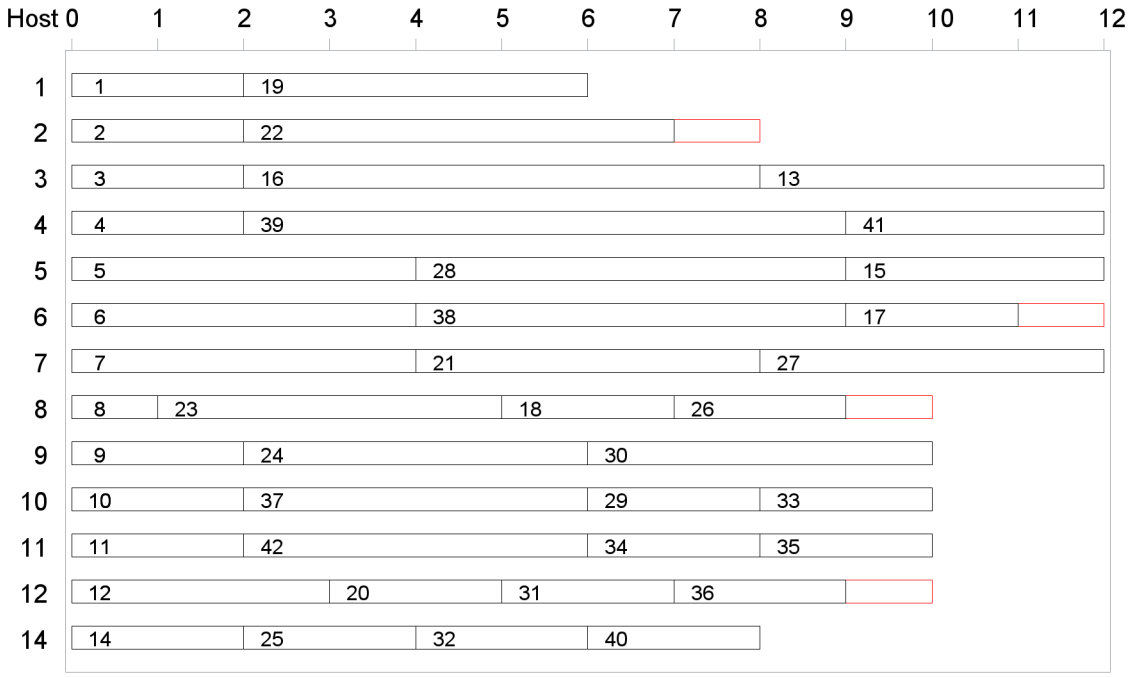
The two charts in Output 6.9.2 show the boat assignments for the first two rounds. The horizontal axis shows the load for each host boat. Slack capacity is highlighted in red.

Output 6.9.2 Gantt Chart: Boat Schedule by Round



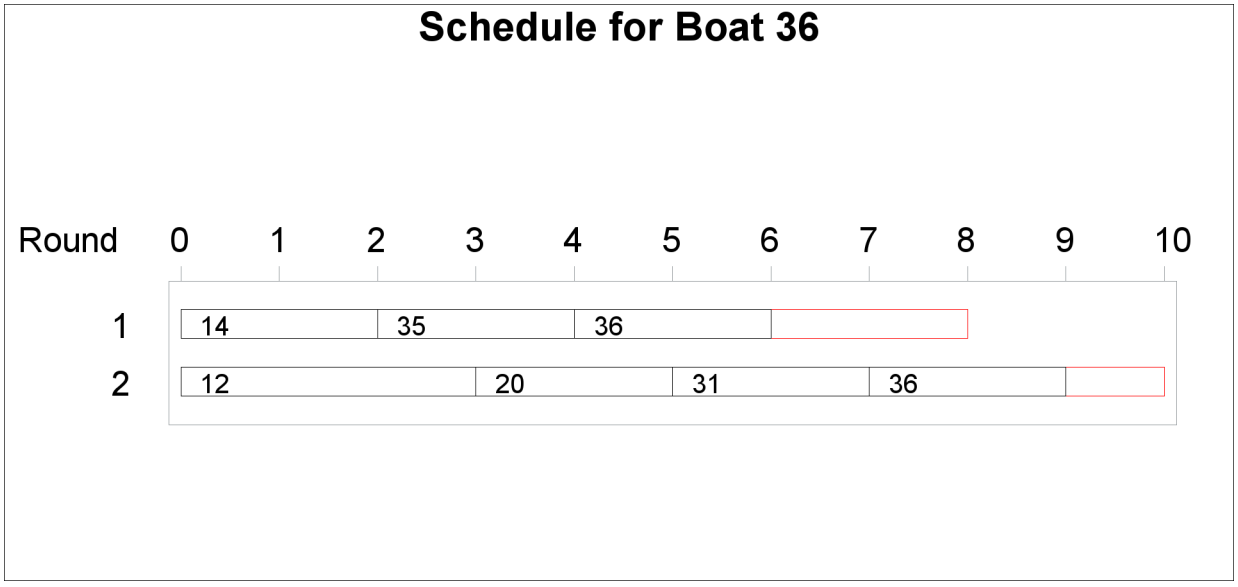
Output 6.9.2 *continued*

Schedule for Round 2

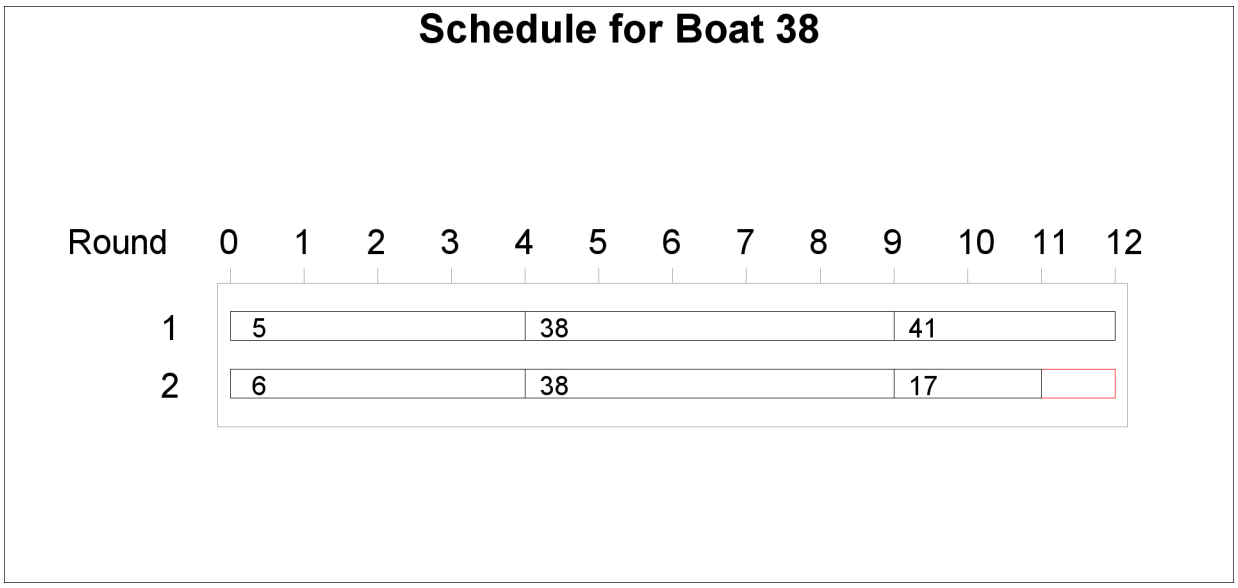


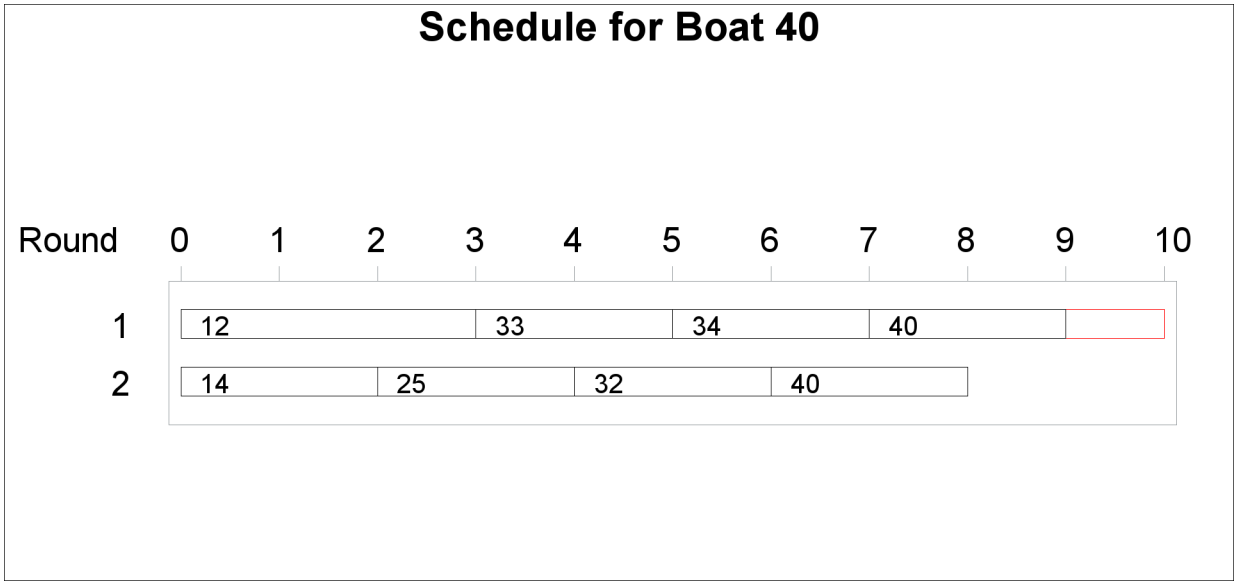
The charts in Output 6.9.3 break down the assignments by boat number for selected boats.

Output 6.9.3 Gantt Chart: Host Boat Schedule by Round

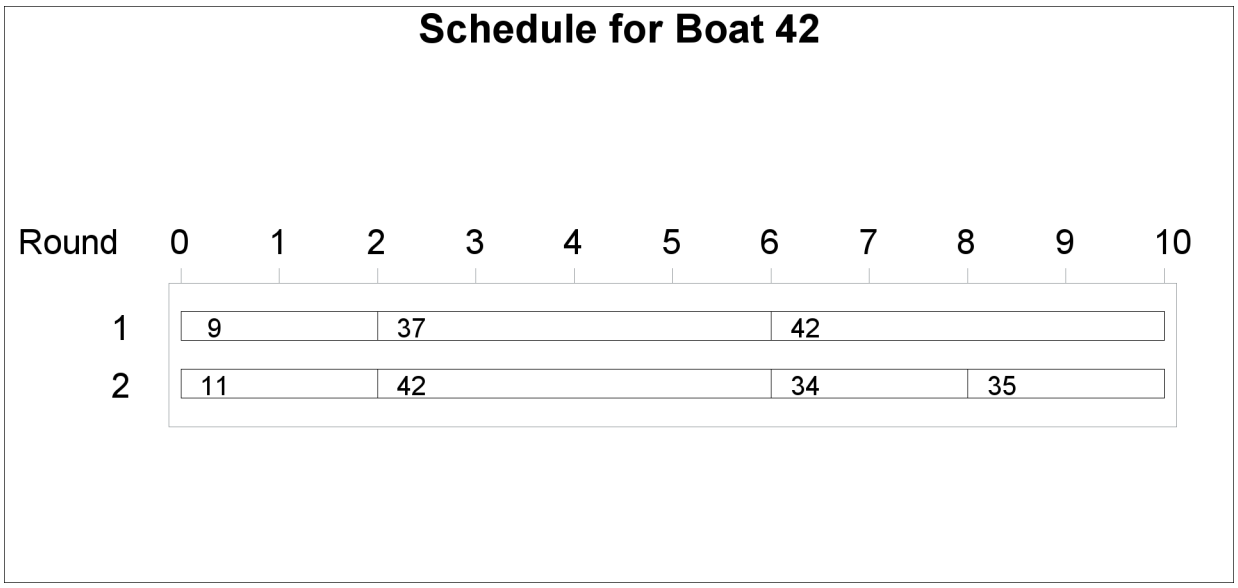


Output 6.9.3 *continued*





Output 6.9.3 *continued*



References

Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004). "Boosting Systematic Search by Weighting Constraints." In *ECAI 2004: Proceedings of the Sixteenth European Conference on Artificial Intelligence*, 146–150. Amsterdam: IOS Press.

Brualdi, R. A. (2010). *Introductory Combinatorics*. 5th ed. Englewood Cliffs, NJ: Prentice-Hall.

Dincbas, M., Simonis, H., and Van Hentenryck, P. (1988). "Solving the Car-Sequencing Problem in Constraint Logic Programming." In *Proceedings of the European Conference on Artificial Intelligence, ECAI-88*, edited by Y. Kodratoff, 290–295. London: Pitman.

Floyd, R. W. (1967). "Nondeterministic Algorithms." *Journal of the ACM* 14:636–644.

Frisch, A. M., Hnich, B., Kiziltan, Z., Miguel, I., and Walsh, T. (2002). "Global Constraints for Lexicographic Orderings." In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, edited by P. Van Hentenryck, 93–108. London: Springer-Verlag.

Garey, M. R., and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman.

Gravel, M., Gagne, C., and Price, W. L. (2005). "Review and Comparison of Three Methods for the Solution of the Car Sequencing Problem." *Journal of the Operational Research Society* 56:1287–1295.

Haralick, R. M., and Elliott, G. L. (1980). "Increasing Tree Search Efficiency for Constraint Satisfaction Problems." *Artificial Intelligence* 14:263–313.

Kumar, V. (1992). "Algorithms for Constraint-Satisfaction Problems: A Survey." *AI Magazine* 13:32–44.

Mackworth, A. K. (1977). "Consistency in Networks of Relations." *Artificial Intelligence* 8:99–118.

Meseguer, P., and Torras, C. (2001). "Exploiting Symmetries within Constraint Satisfaction Search." *Artificial Intelligence* 129:133–163.

Prestwich, S. D. (2001). "Balanced Incomplete Block Design as Satisfiability." In *Twelfth Irish Conference on Artificial Intelligence and Cognitive Science*, 189–198. Maynooth: National University of Ireland.

Riley, P., and Taalman, L. (2008). "Brainfreeze Puzzles." http://www.geekhaus.com/brainfreeze/piday2008.html.

Smith, B. M., Brailsford, S. C., Hubbard, P. M., and Williams, H. P. (1996). "The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared." *Constraints* 1:119–138.

Sokol, J. (2002). *Modeling Automobile Paint Blocking: A Time Window Traveling Salesman Problem*. Ph.D. diss., Massachusetts Institute of Technology.

Solnon, C., Cung, V. D., Nguyen, A., and Artigues, C. (2008). "The Car Sequencing Problem: Overview of State-of-the-Art Methods and Industrial Case-Study of the ROADEF 2005 Challenge Problem." *European Journal of Operational Research* 191:912–927.

Trick, M. A. (2004). "Constraint Programming: A Tutorial." http://mat.gsia.cmu.edu/trick/cp.ppt.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*. London: Academic Press.

Van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming*. Cambridge, MA: MIT Press.

Van Hentenryck, P. (2002). "Constraint and Integer Programming in OPL." *INFORMS Journal on Computing* 14:345–372.

Waltz, D. L. (1975). "Understanding Line Drawings of Scenes with Shadows." In *The Psychology of Computer Vision*, edited by P. H. Winston, 19–91. New York: McGraw-Hill.

Chapter 7
The Linear Programming Solver

Contents

Overview: LP Solver **256**
Getting Started: LP Solver **256**
Syntax: LP Solver **259**
Functional Summary 259
LP Solver Options 260
Details: LP Solver **266**
Presolve 266
Pricing Strategies for the Primal and Dual Simplex Algorithms 266
The Network Simplex Algorithm 266
The Interior Point Algorithm 267
Iteration Log for the Primal and Dual Simplex Algorithms 269
Iteration Log for the Network Simplex Algorithm 270
Iteration Log for the Interior Point Algorithm 271
Iteration Log for the Crossover Algorithm 271
Concurrent LP 272
Parallel Processing 272
Problem Statistics 272
Variable and Constraint Status 273
Irreducible Infeasible Set 274
Macro Variable _OROPTMODEL_ 275
Examples: LP Solver **278**
Example 7.1: Diet Problem 278
Example 7.2: Reoptimizing the Diet Problem Using BASIS=WARMSTART 280
Example 7.3: Two-Person Zero-Sum Game 289
Example 7.4: Finding an Irreducible Infeasible Set 292
Example 7.5: Using the Network Simplex Algorithm 295
Example 7.6: Migration to OPTMODEL: Generalized Networks 303
Example 7.7: Migration to OPTMODEL: Maximum Flow 307
Example 7.8: Migration to OPTMODEL: Production, Inventory, Distribution 310
Example 7.9: Migration to OPTMODEL: Shortest Path 318
References **321**

Overview: LP Solver

The OPTMODEL procedure provides a framework for specifying and solving linear programs (LPs). A standard linear program has the following formulation:

$$\begin{aligned} \min \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \; \{\geq, =, \leq\} \; \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of decision variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the matrix of constraints
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^n$	is the vector of lower bounds on variables
$\mathbf{u}$	$\in$	$\mathbb{R}^n$	is the vector of upper bounds on variables

The following LP algorithms are available in the OPTMODEL procedure:

- primal simplex algorithm
- dual simplex algorithm
- network simplex algorithm
- interior point algorithm

The primal and dual simplex algorithms implement the two-phase simplex method. In phase I, the algorithm tries to find a feasible solution. If no feasible solution is found, the LP is infeasible; otherwise, the algorithm enters phase II to solve the original LP. The network simplex algorithm extracts a network substructure, solves this using network simplex, and then constructs an advanced basis to feed to either primal or dual simplex. The interior point algorithm implements a primal-dual predictor-corrector interior point algorithm. If any of the decision variables are constrained to be integer-valued, then the relaxed version of the problem is solved.

Getting Started: LP Solver

The following example illustrates how you can use the OPTMODEL procedure to solve linear programs. Suppose you want to solve the following problem:

$$\begin{array}{rlllllll} \max & x_1 & + & x_2 & + & x_3 & & \\ \text{subject to} & 3x_1 & + & 2x_2 & - & x_3 & \leq & 1 \\ & -2x_1 & - & 3x_2 & + & 2x_3 & \leq & 1 \\ & & & x_1, & & x_2, \quad x_3 & \geq & 0 \end{array}$$

You can use the following statements to call the OPTMODEL procedure for solving linear programs:

```
proc optmodel;
   var x{i in 1..3} >= 0;
   max f =    x[1] +   x[2] +   x[3];
   con c1:  3*x[1] + 2*x[2] -   x[3] <= 1;
   con c2: -2*x[1] - 3*x[2] + 2*x[3] <= 1;
   solve with lp / algorithm = ps presolver = none logfreq = 1;
   print x;
quit;
```

The optimal solution and the optimal objective value are displayed in Figure 7.1.

Figure 7.1 Solution Summary

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Maximization
Objective Function	f
Objective Type	Linear
Number of Variables	3
Bounded Above	0
Bounded Below	3
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	2
Linear LE (<=)	2
Linear EQ (=)	0
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	6

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Figure 7.1 *continued*

Solution Summary	
Solver	LP
Algorithm	Primal Simplex
Objective Function	f
Solution Status	Optimal
Objective Value	8
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	4
Presolve Time	0.00
Solution Time	0.00

[1]	x
1	0
2	3
3	5

The iteration log displaying problem statistics, progress of the solution, and the optimal objective value is shown in Figure 7.2.

Figure 7.2 Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 2 linear constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.
                          Objective                Entering      Leaving
      Phase Iteration       Value         Time     Variable      Variable
       P 1          1    0.000000E+00          0
       P 2          2    0.000000E+00          0       x[3]             c2 (S)
       P 2          3    5.000000E-01          0       x[2]             c1 (S)
       P 2          4    8.000000E+00          0
NOTE: Optimal.
NOTE: Objective = 8.
NOTE: The Primal Simplex solve time is 0.00 seconds.
```

Syntax: LP Solver

The following statement is available in the OPTMODEL procedure:

SOLVE WITH LP < / *options* > **;**

Functional Summary

Table 7.1 summarizes the list of options available for the SOLVE WITH LP statement, classified by function.

Table 7.1 Options for the LP Solver

Description	Option
Solver Options	
Specifies the type of algorithm	ALGORITHM=
Specifies the type of algorithm called after network simplex	ALGORITHM2=
Enables or disables IIS detection	IIS=
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Controls the dualization of the problem	DUALIZE=
Control Options	
Specifies the feasibility tolerance	FEASTOL=
Specifies the frequency of printing solution progress	LOGFREQ=
Specifies the detail of solution progress printed in log	LOGLEVEL=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the optimality tolerance	OPTTOL=
Specifies units of CPU time or real time	TIMETYPE=
Simplex Algorithm Options	
Specifies the type of initial basis	BASIS=
Specifies the type of pricing strategy	PRICETYPE=
Specifies the queue size for determining entering variable	QUEUESIZE=
Enables or disables scaling of the problem	SCALE=
Specifies the initial seed for the random number generator	SEED=
Interior Point Algorithm Options	
Enables or disables interior crossover	CROSSOVER=
Specifies the stopping criterion based on duality gap	STOP_DG=
Specifies the stopping criterion based on dual infeasibility	STOP_DI=
Specifies the stopping criterion based on primal infeasibility	STOP_PI=

Table 7.1 (continued)

Description	Option
Decomposition Algorithm Options	
Enables decomposition algorithm and specifies general control options	DECOMP=()
Specifies options for the master problem	DECOMP_MASTER=()
Specifies options for the subproblem	DECOMP_SUBPROB=()

LP Solver Options

This section describes the options recognized by the LP solver. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the LP solver is explicitly specified using a WITH clause.

If the LP solver terminates before reaching an optimal solution, an intermediate solution is available. You can access this solution by using the .sol variable suffix in the OPTMODEL procedure. See the section "Suffixes" on page 134 for details.

Solver Options

IIS=*number* | *string*

specifies whether the LP solver attempts to identify a set of constraints and variables that form an irreducible infeasible set (IIS). Table 7.2 describes the valid values of the IIS= option.

Table 7.2 Values for IIS= Option

number	*string*	**Description**
0	OFF	Disables IIS detection.
1	ON	Enables IIS detection.

If an IIS is found, information about the infeasibilities can be found in the .status values of the constraints and variables. The default value of this option is OFF. See the section "Irreducible Infeasible Set" on page 274 for details about the IIS= option. See "Suffixes" on page 134 for details about the .status suffix.

ALGORITHM=*option*

SOLVER=*option*

SOL=*option*

specifies one of the following LP algorithms:

Option	Description
PRIMAL (PS)	Uses primal simplex algorithm.
DUAL (DS)	Uses dual simplex algorithm.
NETWORK (NS)	Uses network simplex algorithm.
INTERIORPOINT (IP)	Uses interior point algorithm.
CONCURRENT (CON)	Uses several different algorithms in parallel.

The valid abbreviated value for each option is indicated in parentheses. By default, the dual simplex algorithm is used.

ALGORITHM2=*option*

SOLVER2=*option*

specifies one of the following LP algorithms if ALGORITHM=NS:

Option	Description
PRIMAL (PS)	Uses primal simplex algorithm (after network simplex).
DUAL (DS)	Uses dual simplex algorithm (after network simplex).

The valid abbreviated value for each option is indicated in parentheses. By default, the LP solver decides which algorithm is best to use after calling the network simplex algorithm on the extracted network.

Presolve Options

PRESOLVER=*number* | *string*

specifies one of the following presolve options:

number	*string*	Description
–1	AUTOMATIC	Applies presolver by using default settings.
0	NONE	Disables the presolver.
1	BASIC	Performs basic presolve such as removing empty rows, columns, and fixed variables.
2	MODERATE	Performs basic presolve and applies other inexpensive presolve techniques.
3	AGGRESSIVE	Performs moderate presolve and applies other aggressive (but expensive) presolve techniques.

The default option is AUTOMATIC. See the section "Presolve" on page 266 for details.

DUALIZE=*number* | *string*

controls the dualization of the problem:

number	*string*	Description
–1	AUTOMATIC	The presolver uses a heuristic to decide whether to dualize the problem or not.
0	OFF	Disables dualization. The optimization problem is solved in the form that you specify.
1	ON	The presolver formulates the dual of the linear optimization problem.

Dualization is usually helpful for problems that have many more constraints than variables. You can use this option with all simplex algorithms in the SOLVE WITH LP statement, but it is most effective with the primal and dual simplex algorithms.

The default option is AUTOMATIC.

Control Options

FEASTOL=ϵ

specifies the feasibility tolerance, $\epsilon \in$[1E–9, 1E–4], for determining the feasibility of a variable. The default value is 1E–6.

LOGFREQ=*k*

PRINTFREQ=*k*

specifies that the printing of the solution progress to the iteration log is to occur after every *k* iterations. The print frequency, *k*, is an integer between zero and the largest four-byte signed integer, which is $2^{31} - 1$.

The value $k = 0$ disables the printing of the progress of the solution. If the primal or dual simplex algorithms are used, the default value of this option is determined dynamically according to the problem size. If the network simplex algorithm is used, the default value of this option is 10,000. If the interior point algorithm is used, the default value of this option is 1.

LOGLEVEL=*number* | *string*

PRINTLEVEL2=*number* | *string*

controls the amount of information displayed in the SAS log by the LP solver, from a short description of presolve information and summary to details at each iteration. Table 7.7 describes the valid values for this option.

Table 7.7 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all solver-related messages to SAS log.
1	BASIC	Displays a solver summary after stopping.
2	MODERATE	Prints a solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.
3	AGGRESSIVE	Prints a detailed solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.

The default value is MODERATE.

MAXITER=*k*

specifies the maximum number of iterations. The value *k* can be any integer between one and the largest four-byte signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed. For network simplex, this iteration limit corresponds to the algorithm called after network simplex (either primal or dual simplex).

MAXTIME=*t*

specifies an upper limit of *t* units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OPTTOL=ϵ

specifies the optimality tolerance, $\epsilon \in$ [1E–9, 1E–4], for declaring optimality. The default value is 1E–6.

TIMETYPE=*number* | *string*

specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. Table 7.8 describes the valid values of the TIMETYPE= option.

Table 7.8 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time.
1	REAL	Specifies units of real time.

The "Optimization Statistics" table, an output of the OPTMODEL procedure if you specify PRINTLEVEL=2 in the PROC OPTMODEL statement, also includes the same time units for Presolver Time and Solver Time. The other times (such as Problem Generation Time) in the "Optimization Statistics" table are also in the same units.

The default value of the TIMETYPE= option depends on the algorithm used and on various options. When the solver is used with distributed or multithreaded processing, then by default TIMETYPE= REAL. Otherwise, by default TIMETYPE= CPU. Table 7.9 describes the detailed logic for determining the default; the first context in the table that applies determines the default value. The NTHREADS= and NODES= options are specified in the PERFORMANCE statement of the OPTMODEL procedure. For more information about the NTHREADS= and NODES= options, see the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Table 7.9 Default Value for TIMETYPE= Option

Context	**Default**
Solver is invoked in an OPTMODEL COFOR loop	REAL
NODES= value is nonzero for the decomposition algorithm	REAL
NTHREADS= value is greater than 1 and NODES=0 for the decomposition algorithm	REAL
NTHREADS= value is greater than 1 and ALGORITHM=IP or ALGORITHM=CON	REAL
Otherwise	CPU

Simplex Algorithm Options

BASIS=*number* | *string*

specifies the following options for generating an initial basis:

number	*string*	**Description**
0	CRASH	Generate an initial basis by using crash techniques (Maros 2003). The procedure creates a triangular basic matrix consisting of both decision variables and slack variables.
1	SLACK	Generate an initial basis by using all slack variables.
2	WARMSTART	Start the primal and dual simplex algorithms with the available basis.

The default option is determined automatically based on the problem structure. For network simplex, this option has no effect.

PRICETYPE=*number* | *string*

specifies one of the following pricing strategies for the primal and dual simplex algorithms:

number	*string*	**Description**
0	HYBRID	Use hybrid Devex and steepest-edge pricing strategies. Available for primal simplex algorithm only.
1	PARTIAL	Use partial pricing strategy. Optionally, you can specify QUEUESIZE=. Available for primal simplex algorithm only.
2	FULL	Use the most negative reduced cost pricing strategy.
3	DEVEX	Use Devex pricing strategy.
4	STEEPESTEDGE	Use steepest-edge pricing strategy.

The default option is determined automatically based on the problem structure. For the network simplex algorithm, this option applies only to the algorithm specified by the ALGORITHM2= option. See the section "Pricing Strategies for the Primal and Dual Simplex Algorithms" on page 266 for details.

QUEUESIZE=*k*

specifies the queue size, $k \in [1, n]$, where n is the number of decision variables. This queue is used for finding an entering variable in the simplex iteration. The default value is chosen adaptively based on the number of decision variables. This option is used only when PRICETYPE=PARTIAL.

SCALE=*number* | *string*

specifies one of the following scaling options:

number	*string*	**Description**
0	NONE	Disable scaling.
–1	AUTOMATIC	Automatically apply scaling procedure if necessary.

The default option is AUTOMATIC.

SEED=*number*

specifies the initial seed for the random number generator. Because the seed affects the perturbation in the simplex algorithms, the result might be a different optimal solution and a different solver path, but the effect is usually negligible. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. By default, SEED=100.

Interior Point Algorithm Options

CROSSOVER=*number* | *string*

specifies whether to convert the interior point solution to a basic simplex solution. The values of this option are:

number	*string*	**Description**
0	OFF	Disable crossover.
1	ON	Apply the crossover algorithm to the interior point solution.

If the interior point algorithm terminates with a solution, the crossover algorithm uses the interior point solution to create an initial basic solution. After performing primal fixing and dual fixing, the crossover algorithm calls a simplex algorithm to locate an optimal basic solution. The default value of the CROSSOVER= option is ON.

STOP_DG=δ

specifies the desired relative duality gap, $\delta \in$[1E–9, 1E–4]. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 267 for details.

STOP_DI=β

specifies the maximum allowed relative dual constraints violation, $\beta \in$ [1E–9, 1E–4]. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 267 for details.

STOP_PI=α

specifies the maximum allowed relative bound and primal constraints violation, $\alpha \in$[1E–9, 1E–4]. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 267 for details.

Decomposition Algorithm Options

The following options are available for the decomposition algorithm in the LP solver. For information about the decomposition algorithm, see Chapter 15, "The Decomposition Algorithm."

DECOMP=(*options***)**

enables the decomposition algorithm and specifies overall control options for the algorithm. For more information about this option, see Chapter 15, "The Decomposition Algorithm."

DECOMP_MASTER=(*options***)**

specifies options for the master problem. For more information about this option, see Chapter 15, "The Decomposition Algorithm."

DECOMP_SUBPROB=(*options*)
specifies option for the subproblem. For more information about this option, see Chapter 15, "The Decomposition Algorithm."

Details: LP Solver

Presolve

Presolve in the simplex LP algorithms of PROC OPTMODEL uses a variety of techniques to reduce the problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels or disable it by specifying the PRESOLVER= option.

Pricing Strategies for the Primal and Dual Simplex Algorithms

Several pricing strategies for the primal and dual simplex algorithms are available. Pricing strategies determine which variable enters the basis at each simplex pivot. These can be controlled by specifying the PRICETYPE= option.

The primal simplex algorithm has the following five pricing strategies:

PARTIAL	scans a queue of decision variables to find an entering variable. You can optionally specify the QUEUESIZE= option to control the length of this queue.
FULL	uses Dantzig's most violated reduced cost rule (Dantzig 1963). It compares the reduced cost of all decision variables, and selects the variable with the most violated reduced cost as the entering variable.
DEVEX	implements the Devex pricing strategy developed by Harris (1973).
STEEPESTEDGE	uses the steepest-edge pricing strategy developed by Forrest and Goldfarb (1992).
HYBRID	uses a hybrid of the Devex and steepest-edge pricing strategies.

The dual simplex algorithm has only three pricing strategies available: FULL, DEVEX, and STEEPEST-EDGE.

The Network Simplex Algorithm

The network simplex algorithm in PROC OPTMODEL attempts to leverage the speed of the network simplex algorithm to more efficiently solve linear programs by using the following process:

1. It heuristically extracts the largest possible network substructure from the original problem.
2. It uses the network simplex algorithm to solve for an optimal solution to this substructure.
3. It uses this solution to construct an advanced basis to warm-start either the primal or dual simplex algorithm on the original linear programming problem.

The network simplex algorithm is a specialized version of the simplex algorithm that uses spanning-tree bases to more efficiently solve linear programming problems that have a pure network form. Such LPs can be modeled using a formulation over a directed graph, as a minimum-cost flow problem. Let $G = (N, A)$ be a directed graph, where N denotes the nodes and A denotes the arcs of the graph. The decision variable x_{ij} denotes the amount of flow sent from node i to node j. The cost per unit of flow on the arcs is designated by c_{ij}, and the amount of flow sent across each arc is bounded to be within $[l_{ij}, u_{ij}]$. The demand (or supply) at each node is designated as b_i, where $b_i > 0$ denotes a supply node and $b_i < 0$ denotes a demand node. The corresponding linear programming problem is as follows:

$$\begin{array}{rlcll} \min & \sum_{(i,j)\in A} c_{ij} x_{ij} & & & \\ \text{subject to} & \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} & = & b_i & \forall i \in N \\ & x_{ij} & \leq & u_{ij} & \forall (i,j) \in A \\ & x_{ij} & \geq & l_{ij} & \forall (i,j) \in A. \end{array}$$

The network simplex algorithm used in PROC OPTMODEL is the primal network simplex algorithm. This algorithm finds the optimal primal feasible solution and a dual solution that satisfies complementary slackness. Sometimes the directed graph G is disconnected. In this case, the problem can be decomposed into its weakly connected components, and each minimum-cost flow problem can be solved separately. After solving each component, the optimal basis for the network substructure is augmented with the non-network variables and constraints from the original problem. This advanced basis is then used as a starting point for the primal or dual simplex method. The solver automatically selects the algorithm to use after network simplex. However, you can override this selection with the ALGORITHM2= option.

The network simplex algorithm can be more efficient than the other algorithms on problems that have a large network substructure. The size of this network structure can be seen in the log.

The Interior Point Algorithm

The interior point LP algorithm in PROC OPTMODEL implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following LP formulation (the primal):

$$\begin{array}{rl} \min & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

The corresponding dual is as follows:

$$\begin{array}{rlcll} \max & \mathbf{b}^{\mathrm{T}}\mathbf{y} & & & \\ \text{subject to} & \mathbf{A}^{\mathrm{T}}\mathbf{y} + \mathbf{w} & = & \mathbf{c} & \\ & \mathbf{y} & \geq & \mathbf{0} & \\ & \mathbf{w} & \geq & \mathbf{0} & \end{array}$$

where $\mathbf{y} \in \mathbb{R}^m$ refers to the vector of dual variables and $\mathbf{w} \in \mathbb{R}^n$ refers to the vector of dual slack variables.

The dual makes an important contribution to the certificate of optimality for the primal. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows:

$$\begin{array}{rcll} \mathbf{Ax} - \mathbf{s} & = & \mathbf{b} & \text{(Primal Feasibility)} \\ \mathbf{A}^{\mathrm{T}}\mathbf{y} + \mathbf{w} & = & \mathbf{c} & \text{(Dual Feasibility)} \\ \mathbf{WXe} & = & \mathbf{0} & \text{(Complementarity)} \\ \mathbf{SYe} & = & \mathbf{0} & \text{(Complementarity)} \\ \mathbf{x},\ \mathbf{y},\ \mathbf{w},\ \mathbf{s} & \geq & \mathbf{0} & \end{array}$$

where $\mathbf{e} \equiv (1, \ldots, 1)^{\mathrm{T}}$ of appropriate dimension and $\mathbf{s} \in \mathbb{R}^m$ is the vector of primal *slack* variables.

NOTE: Slack variables (the *s* vector) are automatically introduced by the algorithm when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the algorithm to handle slack variables much more efficiently.

The letters **X**, **Y**, **W**, and **S** denote matrices with corresponding *x*, *y*, *w*, and *s* on the main diagonal and zero elsewhere, as in the following example:

$$\mathbf{X} \equiv \begin{bmatrix} x_1 & 0 & \cdots & 0 \\ 0 & x_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x_n \end{bmatrix}$$

If $(\mathbf{x}^*, \mathbf{y}^*, \mathbf{w}^*, \mathbf{s}^*)$ is a solution of the previously defined system of equations representing the KKT conditions, then $\mathbf{x}^*$ is also an optimal solution to the original LP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations as follows:

$$\begin{bmatrix} \mathbf{Y}^{-1}\mathbf{S} & \mathbf{A} \\ \mathbf{A}^{\mathrm{T}} & -\mathbf{X}^{-1}\mathbf{W} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{y} \\ \Delta\mathbf{x} \end{bmatrix} = \begin{bmatrix} \Xi \\ \Theta \end{bmatrix}$$

where $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ denote the vector of *search directions* in the primal and dual spaces, respectively; Θ and Ξ constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. The interior point algorithm uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point algorithm is that it takes full advantage of the sparsity in the constraint matrix, thereby enabling it to efficiently solve large-scale linear programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore it is of interest to observe the following four measures:

- Relative primal infeasibility measure α:

$$\alpha = \frac{\|\mathbf{Ax} - \mathbf{b} - \mathbf{s}\|_2}{\|\mathbf{b}\|_2 + 1}$$

- Relative dual infeasibility measure β:

$$\beta = \frac{\|\mathbf{c} - \mathbf{A}^{\mathrm{T}}\mathbf{y} - \mathbf{w}\|_2}{\|\mathbf{c}\|_2 + 1}$$

- Relative duality gap δ:

$$\delta = \frac{|\mathbf{c}^{\mathrm{T}}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{y}|}{|\mathbf{c}^{\mathrm{T}}\mathbf{x}| + 1}$$

- Absolute complementarity γ:

$$\gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i$$

where $\|v\|_2$ is the Euclidean norm of the vector v. These measures are displayed in the iteration log.

Iteration Log for the Primal and Dual Simplex Algorithms

The primal and dual simplex algorithms implement a two-phase simplex algorithm. Phase I finds a feasible solution, which phase II improves to an optimal solution.

When LOGFREQ=1, the following information is printed in the iteration log:

Algorithm	indicates which simplex method is running by printing the letter P (primal) or D (dual).
Phase	indicates whether the algorithm is in phase I or phase II of the simplex method.
Iteration	indicates the iteration number.
Objective Value	indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
Time	indicates the time elapsed (in seconds).
Entering Variable	indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by "(S)". If the entering nonbasic variable has distinct and finite lower and upper bounds, then a "bound swap" can take place in the primal simplex method.
Leaving Variable	indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by "(S)". The leaving variable is the same as the entering variable if a bound swap has taken place.

When you omit the LOGFREQ= option or specify a value larger than 1, only the algorithm, phase, iteration, objective value, and time information is printed in the iteration log.

The behavior of objective values in the iteration log depends on both the current phase and the chosen algorithm. In phase I, both simplex methods have artificial objective values that decrease to 0 when a feasible solution is found. For the dual simplex method, phase II maintains a dual feasible solution, so a minimization problem has increasing objective values in the iteration log. For the primal simplex method,

phase II maintains a primal feasible solution, so a minimization problem has decreasing objective values in the iteration log.

During the solution process, some elements of the LP model might be perturbed to improve performance. In this case the objective values that are printed correspond to the perturbed problem. After reaching optimality for the perturbed problem, the LP solver solves the original problem by switching from the primal simplex method to the dual simplex method (or from the dual simplex method to the primal simplex method). Because the problem might be perturbed again, this process can result in several changes between the two algorithms.

Iteration Log for the Network Simplex Algorithm

After finding the embedded network and formulating the appropriate relaxation, the network simplex algorithm uses a primal network simplex algorithm. In the case of a connected network, with one (weakly connected) component, the log will show the progress of the simplex algorithm. The following information is displayed in the iteration log:

Iteration	indicates the iteration number.
PrimalObj	indicates the primal objective value of the current solution.
Primal Infeas	indicates the maximum primal infeasibility of the current solution.
Time	indicates the time spent on the current component by network simplex.

The frequency of the simplex iteration log is controlled by the LOGFREQ= option. The default value of the LOGFREQ= option is 10,000.

If the network relaxation is disconnected, the information in the iteration log shows progress at the component level. The following information is displayed in the iteration log:

Component	indicates the component number being processed.
Nodes	indicates the number of nodes in this component.
Arcs	indicates the number of arcs in this component.
Iterations	indicates the number of simplex iterations needed to solve this component.
Time	indicates the time spent so far in network simplex.

The frequency of the component iteration log is controlled by the LOGFREQ= option. In this case, the default value of the LOGFREQ= option is determined by the size of the network.

The LOGLEVEL= option adjusts the amount of detail shown. By default, LOGLEVEL=MODERATE and reports as in the preceding description. If LOGLEVEL=NONE, no information is shown. If LOGLEVEL=BASIC, the only information shown is a summary of the network relaxation and the time spent solving the relaxation. If LOGLEVEL=AGGRESSIVE, in the case of one component, the log displays as in the preceding description; in the case of multiple components, for each component, a separate simplex iteration log is displayed.

Iteration Log for the Interior Point Algorithm

The interior point algorithm implements an infeasible primal-dual predictor-corrector interior point algorithm. The following information is displayed in the iteration log:

Iter	indicates the iteration number
Complement	indicates the (absolute) complementarity
Duality Gap	indicates the (relative) duality gap
Primal Infeas	indicates the (relative) primal infeasibility measure
Bound Infeas	indicates the (relative) bound infeasibility measure
Dual Infeas	indicates the (relative) dual infeasibility measure
Time	indicates the time elapsed (in seconds).

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. If they do not, it can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value specified in the option MAXITER= or MAXTIME=. If the complementarity and/or the duality gap do not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

Iteration Log for the Crossover Algorithm

The crossover algorithm takes an optimal solution from the interior point algorithm and transforms it into an optimal basic solution. The iterations of the crossover algorithm are similar to simplex iterations; this similarity is reflected in the format of the iteration logs.

When LOGFREQ=1, the following information is printed in the iteration log:

Phase	indicates whether the primal crossover (PC) or dual crossover (DC) technique is used.
Iteration	indicates the iteration number.
Objective Value	indicates the total amount by which the superbasic variables are off their bound. This value decreases to 0 as the crossover algorithm progresses.
Time	indicates the time elapsed (in seconds).
Entering Variable	indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by “(S)”.
Leaving Variable	indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by “(S)”.

When you omit the LOGFREQ= option or specify a value greater than 1, only the phase, iteration, objective value, and time information are printed in the iteration log.

After all the superbasic variables have been eliminated, the crossover algorithm continues with regular primal or dual simplex iterations.

Concurrent LP

The ALGORITHM=CON option starts several different linear optimization algorithms in parallel in a single-machine mode. The LP solver automatically determines which algorithms to run and how many threads to assign to each algorithm. If sufficient resources are available, the solver runs all four standard algorithms. When the first algorithm finishes, the LP solver returns the results from that algorithm and terminates any other algorithms that are still running. If you specify a value of DETERMINISTIC for the PARALLELMODE= option in the PERFORMANCE statement in the OPTMODEL procedure, the algorithm for which the results are returned is not necessarily the one that finished first. The LP solver deterministically selects the algorithm for which the results are returned. For more information about the PERFORMANCE statement, see the section "PERFORMANCE Statement" on page 21. Regardless of which mode (deterministic or nondeterministic) is in effect, terminating algorithms that are still running might take a significant amount of time.

During concurrent optimization, the procedure displays the iteration log for the dual simplex algorithm. See the section "Iteration Log for the Primal and Dual Simplex Algorithms" on page 269 for more information about this iteration log. Upon termination, the solver displays the iteration log for the algorithm that finishes first, unless the dual simplex algorithm finishes first. If you specify LOGLEVEL=AGGRESSIVE, the LP solver displays the iteration logs for all algorithms that were run concurrently.

If you specify PRINTLEVEL=2 in the PROC OPTMODEL statement and ALGORITHM=CON in the SOLVE WITH LP statement, the LP solver produces an ODS table called ConcurrentSummary. This table contains a summary of the solution statuses of all algorithms that are run concurrently.

Parallel Processing

The interior point and concurrent LP algorithms can be run in single-machine mode (in single-machine mode, the computation is executed by multiple threads on a single computer). The decomposition algorithm can be run in either single-machine or distributed mode (in distributed mode, the computation is executed on multiple computing nodes in a distributed computing environment).

NOTE: Distributed mode requires SAS High-Performance Optimization.

You can specify options for parallel processing in the PERFORMANCE statement, which is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMODEL procedure causes the ODS table "ProblemStatistics" to be generated when the LP solver is called. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 7.3 demonstrates the contents of the ODS table "ProblemStatistics."

Figure 7.3 ODS Table ProblemStatistics

The OPTMODEL Procedure

Problem Statistics	
Number of Constraint Matrix Nonzeros	6
Maximum Constraint Matrix Coefficient	3
Minimum Constraint Matrix Coefficient	1
Average Constraint Matrix Coefficient	2.1666666667
Number of Objective Nonzeros	3
Maximum Objective Coefficient	1
Minimum Objective Coefficient	1
Average Objective Coefficient	1
Number of RHS Nonzeros	2
Maximum RHS	1
Minimum RHS	1
Average RHS	1
Maximum Number of Nonzeros per Column	2
Minimum Number of Nonzeros per Column	2
Average Number of Nonzeros per Column	2
Maximum Number of Nonzeros per Row	3
Minimum Number of Nonzeros per Row	3
Average Number of Nonzeros per Row	3

Variable and Constraint Status

Upon termination of the LP solver, the .status suffix of each decision variable and constraint stores information about the status of that variable or constraint. For more information about suffixes in the OPTMODEL procedure, see the section "Suffixes" on page 134.

Variable Status

The .status suffix of a decision variable specifies the status of that decision variable. The suffix can take one of the following values:

B basic variable

L nonbasic variable at its lower bound

U nonbasic variable at its upper bound

F free variable

A superbasic variable (a nonbasic variable that has a value strictly between its bounds)

I LP model infeasible (all decision variables have .status equal to I)

For the interior point algorithm with IIS= OFF, .status is blank.

The following values can appear only if IIS= ON. See the section "Irreducible Infeasible Set" on page 274 for details.

I_L the lower bound of the variable is needed for the IIS

I_U the upper bound of the variable is needed for the IIS

I_F both bounds of the variable are needed for the IIS (the variable is fixed or has conflicting bounds)

Constraint Status

The .status suffix of a constraint specifies the status of the slack variable for that constraint. The suffix can take one of the following values:

B basic variable

L nonbasic variable at its lower bound

U nonbasic variable at its upper bound

F free variable

A superbasic variable (a nonbasic variable that has a value strictly between its bounds)

I LP model infeasible (all decision variables have .status equal to I)

The following values can appear only if option IIS= ON. See the section "Irreducible Infeasible Set" on page 274 for details.

I_L the "GE" ($\geq$) condition of the constraint is needed for the IIS

I_U the "LE" ($\leq$) condition of the constraint is needed for the IIS

I_F both conditions of the constraint are needed for the IIS (the constraint is an equality or a range constraint with conflicting bounds)

Irreducible Infeasible Set

For a linear programming problem, an irreducible infeasible set (IIS) is an infeasible subset of constraints and variable bounds that will become feasible if any single constraint or variable bound is removed. It is possible to have more than one IIS in an infeasible LP. Identifying an IIS can help isolate the structural infeasibility in an LP.

The presolver in the LP algorithms can detect infeasibility, but it identifies only the variable bound or constraint that triggers the infeasibility.

The IIS=ON option directs the LP solver to search for an IIS in a specified LP. You should specify the OPTMODEL option PRESOLVER=NONE when you specify IIS=ON; otherwise the IIS results can be incomplete. The LP solver does not apply the LP presolver to the problem during the IIS search. If the LP solver detects an IIS, it updates the .status suffix of the decision variables and constraints, and then it stops. The number of iterations that are reported in the macro variable and the ODS table is the total number of

simplex iterations. This total includes the initial LP solve and all subsequent iterations during the constraint deletion phase.

The IIS= option can add special values to the .status suffixes of variables and constraints. (For more information, see the section "Variable and Constraint Status" on page 273.) For constraints, a status of "I_L", "I_U", or "I_F" indicates that the "GE" ($\geq$), "LE" ($\leq$), or "EQ" (=) constraint, respectively, is part of the IIS. For range constraints, a status of "I_L" or "I_U" indicates that the lower or upper bound, respectively, of the constraint is needed for the IIS, and "I_F" indicates that the bounds in the constraint are conflicting. For variables, a status of "I_L", "I_U", or "I_F" indicates that the lower, upper, or both bounds, respectively, of the variable are needed for the IIS. From this information, you can identify both the names of the constraints (variables) in the IIS and the corresponding bound where infeasibility occurs.

Making any one of the constraints or variable bounds in the IIS nonbinding removes the infeasibility from the IIS. In some cases, changing a right-hand side or bound by a finite amount removes the infeasibility. However, the only way to guarantee removal of the infeasibility is to set the appropriate right-hand side or bound to ∞ or $-\infty$. Because it is possible for an LP to have multiple irreducible infeasible sets, simply removing the infeasibility from one set might not make the entire problem feasible. To make the entire problem feasible, you can specify IIS=ON and rerun the LP solver after removing the infeasibility from an IIS. Repeating this process until the LP solver no longer detects an IIS results in a feasible problem. This approach to infeasibility repair can produce different end problems depending on which right-hand sides and bounds you choose to relax.

The IIS= option in the LP solver uses two different methods to identify an IIS:

1. Based on the result of the initial solve, the *sensitivity filter* removes several constraints and variable bounds immediately while still maintaining infeasibility. This phase is quick and dramatically reduces the size of the IIS.

2. Next, the *deletion filter* removes each remaining constraint and variable bound one by one to check which of them are needed to obtain an infeasible system. This second phase is more time consuming, but it ensures that the IIS set that the LP solver returns is indeed irreducible. The progress of the deletion filter is reported at regular intervals. The sensitivity filter might be called again during the deletion filter to improve performance.

See Example 7.4 for an example that demonstrates the use of the IIS= option in locating and removing infeasibilities.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro called _OROPTMODEL_. This variable contains a character string. After each PROC OROPTMODEL run, you can examine this macro by specifying `%put &_OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable. The various terms of the variable after the LP solver is called are interpreted as follows.

STATUS
indicates the solver status at termination. It can take one of the following values:

OK	The solver terminated normally.
SYNTAX_ERROR	Incorrect syntax was used.
DATA_ERROR	The input data were inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem occurred in reading or writing data.
SEMANTIC_ERROR	An evaluation error, such as an invalid operand type, occurred.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produces the solution data in the macro variable. This term appears only when STATUS=OK. It can take one of the following values:

PS	The primal simplex algorithm produced the solution data.
DS	The dual simplex algorithm produced the solution data.
NS	The network simplex algorithm produced the solution data.
IP	The interior point algorithm produced the solution data.
DECOMP	The decomposition algorithm produced the solution data.

When you run algorithms concurrently (ALGORITHM=CON), this term indicates which algorithm is the first to terminate.

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
CONDITIONAL_OPTIMAL	The solution is optimal, but some infeasibilities (primal, dual or bound) exceed tolerances due to scaling or pre-processing.
FEASIBLE	The problem is feasible.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
BAD_PROBLEM_TYPE	The problem type is unsupported by the solver.
ITERATION_LIMIT_REACHED	The maximum allowable number of iterations was reached.
TIME_LIMIT_REACHED	The solver reached its execution time limit.
FUNCTION_CALL_LIMIT_REACHED	The solver reached its limit on function evaluations.
INTERRUPTED	The solver was interrupted externally.
FAILED	The solver failed to converge, possibly due to numerical issues.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

OBJECTIVE
indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the primal constraints by the primal solution. For the interior point algorithm, this term indicates the relative violation of the primal constraints by the primal solution.

DUAL_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the dual constraints by the dual solution. For the interior point algorithm, this term indicates the relative violation of the dual constraints by the dual solution.

BOUND_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the lower or upper bounds by the primal solution. For the interior point algorithm, this term indicates the relative violation of the lower or upper bounds by the primal solution.

DUALITY_GAP
indicates the (relative) duality gap. This term appears only if the option ALGORITHM=INTERIORPOINT is specified in the SOLVE statement.

COMPLEMENTARITY
indicates the (absolute) complementarity. This term appears only if the option ALGORITHM=INTERIORPOINT is specified in the SOLVE statement.

ITERATIONS
indicates the number of iterations taken to solve the problem. When the network simplex algorithm is used, this term indicates the number of network simplex iterations taken to solve the network relaxation. When crossover is enabled, this term indicates the number of interior point iterations taken to solve the problem.

ITERATIONS2
indicates the number of simplex iterations performed by the secondary algorithm. The network simplex algorithm selects the secondary algorithm automatically unless a value has been specified for the ALGORITHM2= option. When crossover is enabled, the secondary algorithm is selected automatically. This term appears only if the network simplex algorithm is used or if crossover is enabled.

PRESOLVE_TIME
indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

Examples: LP Solver

Example 7.1: Diet Problem

Consider the problem of diet optimization. There are six different foods: bread, milk, cheese, potato, fish, and yogurt. The cost and nutrition values per unit are displayed in Table 7.14.

Table 7.14 Cost and Nutrition Values

	Bread	Milk	Cheese	Potato	Fish	Yogurt
Cost	2.0	3.5	8.0	1.5	11.0	1.0
Protein, g	4.0	8.0	7.0	1.3	8.0	9.2
Fat, g	1.0	5.0	9.0	0.1	7.0	1.0
Carbohydrates, g	15.0	11.7	0.4	22.6	0.0	17.0
Calories	90	120	106	97	130	180

The following SAS code creates the data set fooddata of Table 7.14:

```
data fooddata;
   infile datalines;
   input name $  cost  prot  fat  carb  cal;
   datalines;
Bread   2     4     1    15    90
Milk    3.5   8     5    11.7  120
Cheese  8     7     9    0.4   106
Potato  1.5   1.3   0.1  22.6  97
Fish    11    8     7    0     130
Yogurt  1     9.2   1    17    180
;
```

The objective is to find a minimum-cost diet that contains at least 300 calories, not more than 10 grams of protein, not less than 10 grams of carbohydrates, and not less than 8 grams of fat. In addition, the diet should contain at least 0.5 unit of fish and no more than 1 unit of milk.

You can model the problem and solve it by using PROC OPTMODEL as follows:

```
proc optmodel;
   /* declare index set */
   set<str> FOOD;

   /* declare variables */
   var diet{FOOD} >= 0;

   /* objective function */
```

```
num cost{FOOD};
min f=sum{i in FOOD}cost[i]*diet[i];

/* constraints */
num prot{FOOD};
num fat{FOOD};
num carb{FOOD};
num cal{FOOD};
num min_cal, max_prot, min_carb, min_fat;
con cal_con: sum{i in FOOD}cal[i]*diet[i] >= 300;
con prot_con: sum{i in FOOD}prot[i]*diet[i] <= 10;
con carb_con: sum{i in FOOD}carb[i]*diet[i] >= 10;
con fat_con: sum{i in FOOD}fat[i]*diet[i] >= 8;

/* read parameters */
read data fooddata into FOOD=[name] cost prot fat carb cal;

/* bounds on variables */
diet['Fish'].lb = 0.5;
diet['Milk'].ub = 1.0;

/* solve and print the optimal solution */
solve with lp/logfreq=1; /* print each iteration to log */
print diet;
```

The optimal solution and the optimal objective value are displayed in Output 7.1.1.

Output 7.1.1 Optimal Solution to the Diet Problem

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Linear
Number of Variables	6
Bounded Above	0
Bounded Below	5
Bounded Below and Above	1
Free	0
Fixed	0
Number of Constraints	4
Linear LE (<=)	1
Linear EQ (=)	0
Linear GE (>=)	3
Linear Range	0
Constraint Coefficients	23

Output 7.1.1 *continued*

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	f
Solution Status	Optimal
Objective Value	12.081337881
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	6
Presolve Time	0.00
Solution Time	0.00

[1]	diet
Bread	0.000000
Cheese	0.449499
Fish	0.500000
Milk	0.053599
Potato	1.865168
Yogurt	0.000000

Example 7.2: Reoptimizing the Diet Problem Using BASIS=WARMSTART

After an LP is solved, you might want to change a set of the parameters of the LP and solve the problem again. This can be done efficiently in PROC OPTMODEL. The warm start technique uses the optimal solution of the solved LP as a starting point and solves the modified LP problem faster than it can be solved again from scratch. This example illustrates reoptimizing the diet problem described in Example 7.1.

Assume the optimal solution is found by the SOLVE statement. Instead of quitting the OPTMODEL procedure, you can continue to solve several variations of the original problem.

Suppose the cost of cheese increases from 8 to 10 per unit and the cost of fish decreases from 11 to 7 per serving unit. You can change the parameters and solve the modified problem by submitting the following code:

```
cost['Cheese']=10; cost['Fish']=7;
solve with lp/presolver=none
             basis=warmstart
             algorithm=ps
             logfreq=1;
print diet;
```

Note that the primal simplex algorithm is preferred because the primal solution to the last-solved LP is still feasible for the modified problem in this case. The solutions to the original diet problem and the modified problem are shown in Output 7.2.1.

Output 7.2.1 Optimal Solutions to the Original Diet Problem and the Diet Problem with Modified Objective Function

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Linear
Number of Variables	6
Bounded Above	0
Bounded Below	5
Bounded Below and Above	1
Free	0
Fixed	0
Number of Constraints	4
Linear LE (<=)	1
Linear EQ (=)	0
Linear GE (>=)	3
Linear Range	0
Constraint Coefficients	23

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	f
Solution Status	Optimal
Objective Value	12.081337881
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	6
Presolve Time	0.00
Solution Time	0.00

Output 7.2.1 *continued*

[1]	diet
Bread	0.000000
Cheese	0.449499
Fish	0.500000
Milk	0.053599
Potato	1.865168
Yogurt	0.000000

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Linear
Number of Variables	6
Bounded Above	0
Bounded Below	5
Bounded Below and Above	1
Free	0
Fixed	0
Number of Constraints	4
Linear LE (<=)	1
Linear EQ (=)	0
Linear GE (>=)	3
Linear Range	0
Constraint Coefficients	23

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	LP
Algorithm	Primal Simplex
Objective Function	f
Solution Status	Optimal
Objective Value	10.980335514
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	1
Presolve Time	0.00
Solution Time	0.00

Output 7.2.1 *continued*

[1]	diet
Bread	0.000000
Cheese	0.449499
Fish	0.500000
Milk	0.053599
Potato	1.865168
Yogurt	0.000000

The following iteration log indicates that it takes the LP solver no more iterations to solve the modified problem by using BASIS=WARMSTART, since the optimal solution to the original problem remains optimal after the objective function is changed.

Output 7.2.2 Log

```
NOTE: There were 6 observations read from the data set WORK.FOODDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 6 variables, 4 constraints, and 23 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective
      Phase Iteration        Value         Time
       D 1          1    0.000000E+00         0
       D 2          2    5.500000E+00         0
       D 2          6    1.208134E+01         0
NOTE: Optimal.
NOTE: Objective = 12.081337881.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.
                          Objective                Entering          Leaving
      Phase Iteration        Value         Time     Variable          Variable
       P 2          1    1.098034E+01         0
NOTE: Optimal.
NOTE: Objective = 10.980335514.
NOTE: The Primal Simplex solve time is 0.00 seconds.
```

Next, restore the original coefficients of the objective function and consider the case that you need a diet that supplies at least 150 calories. You can change the parameters and solve the modified problem by submitting the following code:

```
cost['Cheese']=8; cost['Fish']=11;cal_con.lb=150;
solve with lp/presolver=none
             basis=warmstart
             algorithm=ds
             logfreq=1;
print diet;
```

Note that the dual simplex algorithm is preferred because the dual solution to the last-solved LP is still feasible for the modified problem in this case. The solution is shown in Output 7.2.3.

Output 7.2.3 Optimal Solution to the Diet Problem with Modified RHS

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Linear
Number of Variables	6
Bounded Above	0
Bounded Below	5
Bounded Below and Above	1
Free	0
Fixed	0
Number of Constraints	4
Linear LE (<=)	1
Linear EQ (=)	0
Linear GE (>=)	3
Linear Range	0
Constraint Coefficients	23

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	f
Solution Status	Optimal
Objective Value	9.1744131985
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	2
Presolve Time	0.00
Solution Time	0.00

Output 7.2.3 *continued*

[1]	diet
Bread	0.00000
Cheese	0.18481
Fish	0.50000
Milk	0.56440
Potato	0.14702
Yogurt	0.00000

The following iteration log indicates that it takes the LP solver just one more phase II iteration to solve the modified problem by using BASIS=WARMSTART.

Output 7.2.4 Log

```
NOTE: There were 6 observations read from the data set WORK.FOODDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective                Entering          Leaving
      Phase Iteration        Value         Time     Variable          Variable
       D 2          1    8.813205E+00          0        cal_con (S)
      carb_con (S)
       D 2          2    9.174413E+00          0
NOTE: Optimal.
NOTE: Objective = 9.1744131985.
NOTE: The Dual Simplex solve time is 0.00 seconds.
```

Next, restore the original constraint on calories and consider the case that you need a diet that supplies no more than 550 mg of sodium per day. The following row is appended to Table 7.14.

	Bread	Milk	Cheese	Potato	Fish	Yogurt
sodium, mg	148	122	337	186	56	132

You can change the parameters, add the new constraint, and solve the modified problem by submitting the following code:

```
cal_con.lb=300;
num sod{FOOD}=[148 122 337 186 56 132];
con sodium: sum{i in FOOD}sod[i]*diet[i] <= 550;
solve with lp/presolver=none
             basis=warmstart
             logfreq=1;
print diet;
```

The solution is shown in Output 7.2.5.

Output 7.2.5 Optimal Solution to the Diet Problem with Additional Constraint

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Linear
Number of Variables	6
Bounded Above	0
Bounded Below	5
Bounded Below and Above	1
Free	0
Fixed	0
Number of Constraints	5
Linear LE (<=)	2
Linear EQ (=)	0
Linear GE (>=)	3
Linear Range	0
Constraint Coefficients	29

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Output 7.2.5 *continued*

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	f
Solution Status	Optimal
Objective Value	12.081337881
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	1
Presolve Time	0.00
Solution Time	0.00

[1]	diet
Bread	0.000000
Cheese	0.449499
Fish	0.500000
Milk	0.053599
Potato	1.865168
Yogurt	0.000000

The following iteration log indicates that it takes the LP solver no more iterations to solve the modified problem by using the BASIS=WARMSTART option, since the optimal solution to the original problem remains optimal after one more constraint is added.

Output 7.2.6 Log

```
NOTE: There were 6 observations read from the data set WORK.FOODDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 5 linear constraints (2 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 29 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective                Entering          Leaving
      Phase Iteration       Value         Time     Variable          Variable
       D 2          1    1.208134E+01         0
NOTE: Optimal.
NOTE: Objective = 12.081337881.
NOTE: The Dual Simplex solve time is 0.00 seconds.
```

Example 7.3: Two-Person Zero-Sum Game

Consider a two-person zero-sum game (where one person wins what the other person loses). The players make moves simultaneously, and each has a choice of actions. There is a *payoff* matrix that indicates the amount one player gives to the other under each combination of actions:

$$\begin{array}{rc} & \text{Player II plays } j \\ & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \text{Player I plays } i \quad \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} -5 & 3 & 1 & 8 \\ 5 & 5 & 4 & 6 \\ -4 & 6 & 0 & 5 \end{pmatrix} \end{array}$$

If player I makes move i and player II makes move j, then player I wins (and player II loses) a_{ij}. What is the best strategy for the two players to adopt? This example is simple enough to be analyzed from observation. Suppose player I plays 1 or 3; the best response of player II is to play 1. In both cases, player I loses and player II wins. So the best action for player I is to play 2. In this case, the best response for player II is to play 3, which minimizes the loss. In this case, (2, 3) is a *pure-strategy Nash equilibrium* in this game.

For illustration, consider the following mixed strategy case. Assume that player I selects i with probability p_i, $i = 1, 2, 3$, and player II selects j with probability q_j, $j = 1, 2, 3, 4$. Consider player II's problem of minimizing the maximum expected payout:

$$\min_{\mathbf{q}} \left\{ \max_{i} \sum_{j=1}^{4} a_{ij} q_j \right\} \quad \text{subject to} \quad \sum_{j=1}^{4} q_{ij} = 1, \quad \mathbf{q} \geq 0$$

This is equivalent to

$$\begin{aligned} \min_{\mathbf{q},v} \; v \quad \text{subject to} \quad \sum_{j=1}^{4} a_{ij} q_j &\leq v \quad \forall i \\ \sum_{j=1}^{4} q_j &= 1 \\ \mathbf{q} &\geq 0 \end{aligned}$$

The problem can be transformed into a more standard format by making a simple change of variables: $x_j = q_j / v$. The preceding LP formulation now becomes

$$\begin{aligned} \min_{\mathbf{x},v} \; v \quad \text{subject to} \quad \sum_{j=1}^{4} a_{ij} x_j &\leq 1 \quad \forall i \\ \sum_{j=1}^{4} x_j &= 1/v \\ \mathbf{q} &\geq 0 \end{aligned}$$

which is equivalent to

$$\max_{\mathbf{x}} \sum_{j=1}^{4} x_j \quad \text{subject to} \quad A\mathbf{x} \leq \mathbf{1}, \quad \mathbf{x} \geq 0$$

where A is the payoff matrix and $\mathbf{1}$ is a vector of 1's. It turns out that the corresponding optimization problem from player I's perspective can be obtained by solving the dual problem, which can be written as

$$\min_{\mathbf{y}} \sum_{i=1}^{3} y_i \quad \text{subject to} \quad A^{\mathrm{T}}\mathbf{y} \geq \mathbf{1}, \quad \mathbf{y} \geq 0$$

You can model the problem and solve it by using PROC OPTMODEL as follows:

```
proc optmodel;
   num a{1..3, 1..4}=[-5 3 1 8
                       5 5 4 6
                      -4 6 0 5];
   var x{1..4} >= 0;
   max f = sum{i in 1..4}x[i];
   con c{i in 1..3}: sum{j in 1..4}a[i,j]*x[j] <= 1;
   solve with lp / algorithm = ps presolver = none logfreq = 1;
   print x;
   print c.dual;
quit;
```

The optimal solution is displayed in Output 7.3.1.

Output 7.3.1 Optimal Solutions to the Two-Person Zero-Sum Game

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Maximization
Objective Function	f
Objective Type	Linear
Number of Variables	4
Bounded Above	0
Bounded Below	4
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	3
Linear LE (<=)	3
Linear EQ (=)	0
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	11

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	LP
Algorithm	Primal Simplex
Objective Function	f
Solution Status	Optimal
Objective Value	0.25
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	3
Presolve Time	0.00
Solution Time	0.00

[1]	x
1	0.00
2	0.00
3	0.25
4	0.00

Output 7.3.1 *continued*

[1]	c.DUAL
1	0.00
2	0.25
3	0.00

The optimal solution $\mathbf{x}^* = (0, 0, 0.25, 0)$ with an optimal value of 0.25. Therefore the optimal strategy for player II is $\mathbf{q}^* = \mathbf{x}^*/0.25 = (0, 0, 1, 0)$. You can check the optimal solution of the dual problem by using the constraint suffix ".dual". So $\mathbf{y}^* = (0, 0.25, 0)$ and player I's optimal strategy is (0, 1, 0). The solution is consistent with our intuition from observation.

Example 7.4: Finding an Irreducible Infeasible Set

This example demonstrates the use of the IIS= option to locate an irreducible infeasible set. Suppose you want to solve a linear program that has the following simple formulation:

$$
\begin{array}{rrcrcrcrl}
\min & & x_1 & + & x_2 & + & x_3 & & (\text{cost}) \\
\text{subject to} & & x_1 & + & x_2 & & & \geq\ 10 & (\text{con1}) \\
 & & x_1 & & & + & x_3 & \leq\ 4 & (\text{con2}) \\
 & 4 \leq & & & x_2 & + & x_3 & \leq\ 5 & (\text{con3}) \\
 & & & & & x_1, & x_2 & \geq\ 0 & \\
 & & & & 0 & \leq & x_3 & \leq\ 3 &
\end{array}
$$

It is easy to verify that the following three constraints (or rows) and one variable (or column) bound form an IIS for this problem:

$$
\begin{array}{rcrcrcrl}
x_1 & + & x_2 & & & \geq & 10 & (\text{con1}) \\
x_1 & & & + & x_3 & \leq & 4 & (\text{con2}) \\
 & & x_2 & + & x_3 & \leq & 5 & (\text{con3}) \\
 & & & & x_3 & \geq & 0 &
\end{array}
$$

You can formulate the problem and call the LP solver by using the following statements:

```
proc optmodel presolver=none;
   /* declare variables */
   var x{1..3} >=0;

   /* upper bound on variable x[3] */
   x[3].ub = 3;

   /* objective function */
   min obj = x[1] + x[2] + x[3];

   /* constraints */
   con c1: x[1] + x[2] >= 10;
```

```
con c2: x[1] + x[3] <= 4;
con c3: 4 <= x[2] + x[3] <= 5;

solve with lp / iis = on;

print x.status;
print c1.status c2.status c3.status;
```

The notes printed in the log appear in Output 7.4.1.

Output 7.4.1 Finding an IIS: Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 linear constraints (1 LE, 0 EQ, 1 GE, 1 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The IIS= option is enabled.
                          Objective
      Phase Iteration        Value         Time
       P 1          1    1.400000E+01         0
       P 1          3    1.000000E+00         0
NOTE: Applying the IIS sensitivity filter.
NOTE: The sensitivity filter removed 1 constraints and 3 variable bounds.
NOTE: Applying the IIS deletion filter.
NOTE: Processing constraints.
      Processed     Removed      Time
              0           0         0
              1           0         0
              2           0         0
              3           0         0
NOTE: Processing variable bounds.
      Processed     Removed      Time
              0           0         0
              1           0         0
              2           0         0
              3           0         0
NOTE: The deletion filter removed 0 constraints and 0 variable bounds.
NOTE: The IIS= option found this problem to be infeasible.
NOTE: The IIS= option found an irreducible infeasible set with 1 variables and
      3 constraints.
NOTE: The IIS solve time is 0.00 seconds.
```

The output of the PRINT statements appears in Output 7.4.2. The value of the .status suffix for the variables x[1] and x[2] is missing, so the variable bounds for x[1] and x[2] are not in the IIS.

Output 7.4.2 Solution Summary, Variable Status, and Constraint Status

The OPTMODEL Procedure

Solution Summary	
Solver	LP
Algorithm	IIS
Objective Function	obj
Solution Status	Infeasible
Iterations	13
Presolve Time	0.00
Solution Time	0.00

[1]	x.STATUS
1	
2	
3	I_L

c1.STATUS	c2.STATUS	c3.STATUS
I_L	I_U	I_U

The value of c3.status is I_U, which indicates that $x_2 + x_3 \leq 5$ is an element of the IIS. The original constraint is c3, a range constraint with a lower bound of 4. If you choose to remove the constraint $x_2 + x_3 \leq 5$, you can change the value of c3.ub to the largest positive number representable in your operating environment. You can specify this number by using the CONSTANT function.

The modified LP problem is specified and solved by adding the following lines to the original PROC OPTMODEL call.

```
/* relax upper bound on constraint c3 */
c3.ub = constant('BIG');

solve with lp / iis = on;
```

Because one element of the IIS has been removed, the modified LP problem should no longer contain the infeasible set. Due to the size of this problem, there should be no additional irreducible infeasible sets.

The notes shown in Output 7.4.3 are printed to the log.

Output 7.4.3 Infeasibility Removed: Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 linear constraints (1 LE, 0 EQ, 2 GE, 0 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The IIS= option is enabled.
                          Objective
      Phase Iteration       Value         Time
       P 1          1    1.400000E+01         0
       P 1          3    0.000000E+00         0
NOTE: The IIS= option found this problem to be feasible.
NOTE: The IIS solve time is 0.00 seconds.
```

The solution summary and primal solution are displayed in Output 7.4.4.

Output 7.4.4 Infeasibility Removed: Solution

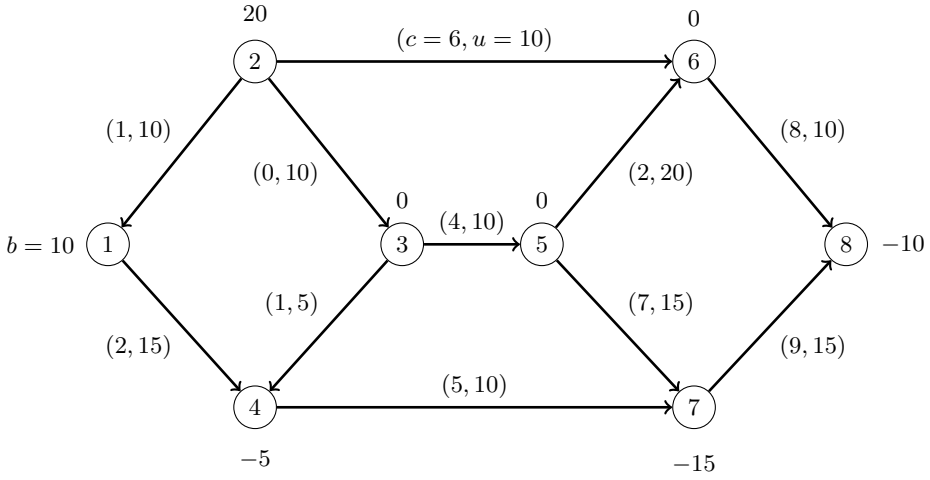
The OPTMODEL Procedure

Solution Summary	
Solver	LP
Algorithm	IIS
Objective Function	obj
Solution Status	Feasible
Iterations	3
Presolve Time	0.00
Solution Time	0.00

Example 7.5: Using the Network Simplex Algorithm

This example demonstrates how you can use the network simplex algorithm to find the minimum-cost flow in a directed graph. Consider the directed graph in Figure 7.4, which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 7.4 Minimum-Cost Network Flow Problem: Data



You can use the following SAS statements to create the input data sets nodedata and arcdata:

```
data nodedata;
   input _node_ $ _sd_;
   datalines;
1    10
2    20
3     0
4    -5
5     0
6     0
7   -15
8   -10
;

data arcdata;
   input _tail_ $ _head_ $ _lo_ _capac_ _cost_;
   datalines;
1    4    0    15    2
2    1    0    10    1
2    3    0    10    0
2    6    0    10    6
3    4    0     5    1
3    5    0    10    4
4    7    0    10    5
5    6    0    20    2
5    7    0    15    7
6    8    0    10    8
7    8    0    15    9
;
```

You can use the following call to PROC OPTMODEL to find the minimum-cost flow:

```
proc optmodel;
   set <str> NODES;
   num supply_demand {NODES};

   set <str,str> ARCS;
   num arcLower  {ARCS};
   num arcUpper  {ARCS};
   num arcCost   {ARCS};

   read data arcdata into ARCS=[_tail_ _head_]
      arcLower=_lo_ arcUpper=_capac_ arcCost=_cost_;
   read data nodedata into NODES=[_node_] supply_demand=_sd_;

   var flow {<i,j> in ARCS} >= arcLower[i,j] <= arcUpper[i,j];
   min obj = sum {<i,j> in ARCS} arcCost[i,j] * flow[i,j];
   con balance {i in NODES}:
      sum {<(i),j> in ARCS} flow[i,j] - sum {<j,(i)> in ARCS} flow[j,i]
         = supply_demand[i];
   solve with lp / algorithm=ns scale=none logfreq=1;
   print flow;
quit;
%put &_OROPTMODEL_;
```

The optimal solution is displayed in Output 7.5.1.

Output 7.5.1 Network Simplex Algorithm: Primal Solution Output

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	obj
Objective Type	Linear
Number of Variables	11
Bounded Above	0
Bounded Below	0
Bounded Below and Above	11
Free	0
Fixed	0
Number of Constraints	8
Linear LE (<=)	0
Linear EQ (=)	8
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	22

Output 7.5.1 *continued*

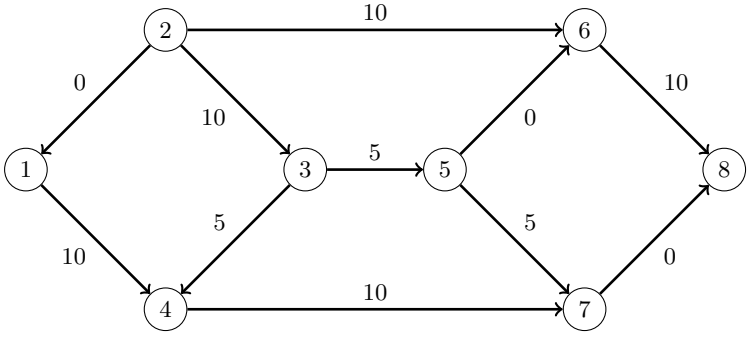
Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

Solution Summary	
Solver	LP
Algorithm	Network Simplex
Objective Function	obj
Solution Status	Optimal
Objective Value	270
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	8
Iterations2	0
Presolve Time	0.00
Solution Time	0.00

[1]	[2]	flow
1	**4**	10
2	**1**	0
2	**3**	10
2	**6**	10
3	**4**	5
3	**5**	5
4	**7**	10
5	**6**	0
5	**7**	5
6	**8**	10
7	**8**	0

The optimal solution is represented graphically in Figure 7.5.

Figure 7.5 Minimum-Cost Network Flow Problem: Optimal Solution



The iteration log is displayed in Output 7.5.2.

Output 7.5.2 Log: Solution Progress

```
NOTE: There were 11 observations read from the data set WORK.ARCDATA.
NOTE: There were 8 observations read from the data set WORK.NODEDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 11 variables (0 free, 0 fixed).
NOTE: The problem has 8 linear constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 22 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem is a pure network instance; PRESOLVER=NONE is used.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Network Simplex algorithm is used.
NOTE: The network has 8 rows (100.00%), 11 columns (100.00%), and 1 component.
NOTE: The network extraction and setup time is 0.00 seconds.
                        Primal         Primal           Dual
      Iteration      Objective  Infeasibility  Infeasibility     Time
              1   0.000000E+00   2.000000E+01   8.900000E+01     0.00
              2   0.000000E+00   2.000000E+01   8.900000E+01     0.00
              3   5.000000E+00   1.500000E+01   8.400000E+01     0.00
              4   5.000000E+00   1.500000E+01   8.300000E+01     0.00
              5   7.500000E+01   1.500000E+01   8.300000E+01     0.00
              6   7.500000E+01   1.500000E+01   7.900000E+01     0.00
              7   1.300000E+02   1.000000E+01   7.600000E+01     0.00
              8   2.700000E+02   0.000000E+00   0.000000E+00     0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.00 seconds.
NOTE: Optimal.
NOTE: Objective = 270.
NOTE: The PROCEDURE OPTMODEL printed pages 1-2.
STATUS=OK ALGORITHM=NS SOLUTION_STATUS=OPTIMAL OBJECTIVE=270
PRIMAL_INFEASIBILITY=0 DUAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0 ITERATIONS=8
ITERATIONS2=0 PRESOLVE_TIME=0.00 SOLUTION_TIME=0.00
```

Now, suppose there is a budget on the flow that comes out of arc 2: the total arc cost of flow that comes out of arc 2 cannot exceed 50. You can use the following call to PROC OPTMODEL to find the minimum-cost flow:

```
proc optmodel;
   set <str> NODES;
   num supply_demand {NODES};

   set <str,str> ARCS;
   num arcLower  {ARCS};
   num arcUpper  {ARCS};
   num arcCost   {ARCS};

   read data arcdata into ARCS=[_tail_ _head_]
```

```
      arcLower=_lo_ arcUpper=_capac_ arcCost=_cost_;
   read data nodedata into NODES=[_node_] supply_demand=_sd_;

   var flow {<i,j> in ARCS} >= arcLower[i,j] <= arcUpper[i,j];
   min obj = sum {<i,j> in ARCS} arcCost[i,j] * flow[i,j];
   con balance {i in NODES}:
      sum {<(i),j> in ARCS} flow[i,j] - sum {<j,(i)> in ARCS} flow[j,i]
         = supply_demand[i];
   con budgetOn2:
      sum {<i,j> in ARCS: i='2'} arcCost[i,j] * flow[i,j] <= 50;
   solve with lp / algorithm=ns scale=none logfreq=1;
   print flow;
quit;
%put &_OROPTMODEL_;
```

The optimal solution is displayed in Output 7.5.3.

Output 7.5.3 Network Simplex Algorithm: Primal Solution Output

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	obj
Objective Type	Linear
Number of Variables	11
Bounded Above	0
Bounded Below	0
Bounded Below and Above	11
Free	0
Fixed	0
Number of Constraints	9
Linear LE (<=)	1
Linear EQ (=)	8
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	24

Performance Information	
Execution Mode	Single-Machine
Number of Threads	1

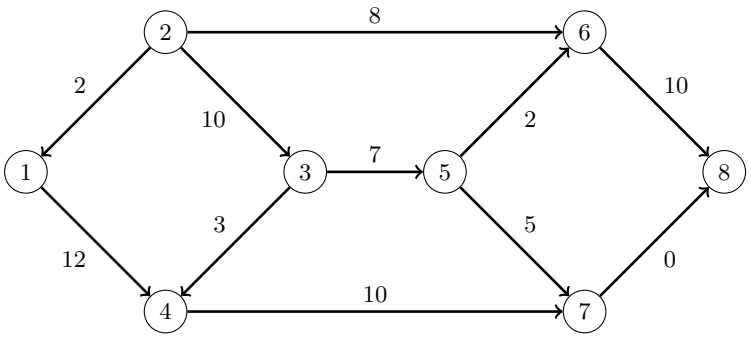
Output 7.5.3 *continued*

Solution Summary	
Solver	LP
Algorithm	Network Simplex
Objective Function	obj
Solution Status	Optimal
Objective Value	274
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	7
Iterations2	2
Presolve Time	0.00
Solution Time	0.00

[1]	[2]	flow
1	**4**	12
2	**1**	2
2	**3**	10
2	**6**	8
3	**4**	3
3	**5**	7
4	**7**	10
5	**6**	2
5	**7**	5
6	**8**	10
7	**8**	0

The optimal solution is represented graphically in Figure 7.6.

Figure 7.6 Minimum-Cost Network Flow Problem: Optimal Solution (with Budget Constraint)



The iteration log is displayed in Output 7.5.4. Note that the network simplex algorithm extracts a subnetwork in this case.

Output 7.5.4 Log: Solution Progress

```
NOTE: There were 11 observations read from the data set WORK.ARCDATA.
NOTE: There were 8 observations read from the data set WORK.NODEDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 11 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (1 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 24 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 4 variables and 4 constraints.
NOTE: The LP presolver removed 7 constraint coefficients.
NOTE: The presolved problem has 7 variables, 5 constraints, and 17 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Network Simplex algorithm is used.
NOTE: The network has 4 rows (80.00%), 7 columns (100.00%), and 1 component.
NOTE: The network extraction and setup time is 0.00 seconds.
                      Primal         Primal           Dual
      Iteration      Objective  Infeasibility  Infeasibility     Time
              1   8.015000E+01   1.006000E+01   5.500000E+01     0.00
              2   1.053000E+02   5.030000E+00   5.400000E+01     0.00
              3   1.053000E+02   5.030000E+00   5.400000E+01     0.00
              4   1.353000E+02   3.000000E-02   4.900000E+01     0.00
              5   1.356300E+02   0.000000E+00   4.700000E+01     0.00
              6   1.356300E+02   0.000000E+00   0.000000E+00     0.00
              7   2.700000E+02   0.000000E+00   0.000000E+00     0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.00 seconds.
NOTE: The Dual Simplex algorithm is used.
                          Objective                Entering           Leaving
      Phase Iteration        Value         Time     Variable           Variable
       D 2          1    2.700000E+02         0   flow['5','6']
      budgetOn2 (S)
       D 2          2    2.740000E+02         0
NOTE: Optimal.
NOTE: Objective = 274.
NOTE: The Simplex solve time is 0.00 seconds.
NOTE: The PROCEDURE OPTMODEL printed pages 4-5.
STATUS=OK ALGORITHM=NS SOLUTION_STATUS=OPTIMAL OBJECTIVE=274
PRIMAL_INFEASIBILITY=0 DUAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0 ITERATIONS=7
ITERATIONS2=2 PRESOLVE_TIME=0.00 SOLUTION_TIME=0.00
```

Example 7.6: Migration to OPTMODEL: Generalized Networks

The following example shows how to use PROC OPTMODEL to solve the example "Generalized Networks: Using the EXCESS= Option" in Chapter 5, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The input data sets are the same as in the PROC NETFLOW example.

```
title 'Generalized Networks';

data garcs;
   input _from_ $ _to_ $ _cost_ _mult_;
   datalines;
s1 d1 1 .
s1 d2 8 .
s2 d1 4 2
s2 d2 2 2
s2 d3 1 2
s3 d2 5 0.5
s3 d3 4 0.5
;

data gnodes;
   input _node_ $ _sd_ ;
   datalines;
s1 5
s2 20
s3 10
d1 -5
d2 -10
d3 -20
;
```

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called GNETOUT:

```
proc optmodel;
   set <str> NODES;
   num _sd_ {NODES} init 0;
   read data gnodes into NODES=[_node_] _sd_;

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS};
   num _mult_ {ARCS} init 1;
   read data garcs nomiss into ARCS=[_from_ _to_] _cost_ _mult_;
   NODES = NODES union (union {<i,j> in ARCS} {i,j});
```

```
   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
   con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} _mult_[j,i] * Flow[j,i] = _sd_[i];

   num infinity = constant('BIG');
   /* change equality constraint to le constraint for supply nodes */
   for {i in NODES: _sd_[i] > 0} balance[i].lb = -infinity;

   solve;

   num _supply_ {<i,j> in ARCS} = (if _sd_[i] ne 0 then _sd_[i] else .);
   num _demand_ {<i,j> in ARCS} = (if _sd_[j] ne 0 then -_sd_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data gnetout from [_from_ _to_]
      _cost_ _capac_ _lo_ _mult_ _supply_ _demand_ _flow_=Flow _fcost_;
quit;
```

To solve a generalized network flow problem, the usual balance constraint is altered to include the arc multiplier "_mult_[i,j]" in the second sum. The balance constraint is initially declared as an equality, but to mimic the EXCESS=SUPPLY option in PROC NETFLOW, the sense of this constraint is changed to "≤" by relaxing the constraint's lower bound for supply nodes. The output data set is displayed in Output 7.6.1.

Output 7.6.1 Optimal Solution with Excess Supply

Obs	_from_	_to_	_cost_	_capac_	_lo_	_mult_	_supply_	_demand_	_flow_	_fcost_
1	s1	d1	1	.	0	1.0	5	5	5	5
2	s1	d2	8	.	0	1.0	5	10	0	0
3	s2	d1	4	.	0	2.0	20	5	0	0
4	s2	d2	2	.	0	2.0	20	10	5	10
5	s2	d3	1	.	0	2.0	20	20	10	10
6	s3	d2	5	.	0	0.5	10	10	0	0
7	s3	d3	4	.	0	0.5	10	20	0	0

The log is displayed in Output 7.6.2.

Output 7.6.2 OPTMODEL Log

```
NOTE: There were 6 observations read from the data set WORK.GNODES.
NOTE: There were 7 observations read from the data set WORK.GARCS.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 7 variables (0 free, 0 fixed).
NOTE: The problem has 6 linear constraints (3 LE, 3 EQ, 0 GE, 0 range).
NOTE: The problem has 14 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 2 variables and 2 constraints.
NOTE: The LP presolver removed 4 constraint coefficients.
NOTE: The presolved problem has 5 variables, 4 constraints, and 10 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective
      Phase Iteration       Value         Time
       D 1          1    0.000000E+00         0
       D 2          2    1.500000E+01         0
       D 2          3    2.500000E+01         0
NOTE: Optimal.
NOTE: Objective = 25.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.GNETOUT has 7 observations and 10 variables.
```

Now consider the previous example but with a slight modification to the arc multipliers, as in the PROC NETFLOW example:

```
data garcs1;
   input _from_ $ _to_ $ _cost_ _mult_;
   datalines;
s1 d1 1 0.5
s1 d2 8 0.5
s2 d1 4 .
s2 d2 2 .
s2 d3 1 .
s3 d2 5 0.5
s3 d3 4 0.5
;
```

The following PROC OPTMODEL statements are identical to the preceding example, except for the balance constraint. The balance constraint is still initially declared as an equality, but to mimic the PROC NETFLOW EXCESS=DEMAND option, the sense of this constraint is changed to "$\geq$" by relaxing the constraint's upper bound for demand nodes.

```
proc optmodel;
   set <str> NODES;
   num _sd_ {NODES} init 0;
   read data gnodes into NODES=[_node_] _sd_;

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS};
   num _mult_ {ARCS} init 1;
   read data garcs1 nomiss into ARCS=[_from_ _to_] _cost_ _mult_;
   NODES = NODES union (union {<i,j> in ARCS} {i,j});

   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
   min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
   con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} _mult_[j,i] * Flow[j,i] = _sd_[i];

   num infinity = constant('BIG');
   /* change equality constraint to ge constraint */
   for {i in NODES: _sd_[i] < 0} balance[i].ub = infinity;

   solve;

   num _supply_ {<i,j> in ARCS} = (if _sd_[i] ne 0 then _sd_[i] else .);
   num _demand_ {<i,j> in ARCS} = (if _sd_[j] ne 0 then -_sd_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data gnetout1 from [_from_ _to_]
      _cost_ _capac_ _lo_ _mult_ _supply_ _demand_ _flow_=Flow _fcost_;
quit;
```

The output data set is displayed in Output 7.6.3.

Output 7.6.3 Optimal Solution with Excess Demand

Obs	_from_	_to_	_cost_	_capac_	_lo_	_mult_	_supply_	_demand_	_flow_	_fcost_
1	s1	d1	1	.	0	0.5	5	5	5	5
2	s1	d2	8	.	0	0.5	5	10	0	0
3	s2	d1	4	.	0	1.0	20	5	0	0
4	s2	d2	2	.	0	1.0	20	10	5	10
5	s2	d3	1	.	0	1.0	20	20	15	15
6	s3	d2	5	.	0	0.5	10	10	0	0
7	s3	d3	4	.	0	0.5	10	20	10	40

The log is displayed in Output 7.6.4.

Output 7.6.4 OPTMODEL Log

```
NOTE: There were 6 observations read from the data set WORK.GNODES.
NOTE: There were 7 observations read from the data set WORK.GARCS1.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 7 variables (0 free, 0 fixed).
NOTE: The problem has 6 linear constraints (0 LE, 3 EQ, 3 GE, 0 range).
NOTE: The problem has 14 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 2 variables and 2 constraints.
NOTE: The LP presolver removed 4 constraint coefficients.
NOTE: The presolved problem has 5 variables, 4 constraints, and 10 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective
      Phase Iteration       Value         Time
       D 1          1    0.000000E+00         0
       D 2          2    4.997000E+01         0
       D 2          4    7.000000E+01         0
NOTE: Optimal.
NOTE: Objective = 70.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.GNETOUT1 has 7 observations and 10 variables.
NOTE: The PROCEDURE OPTMODEL printed pages 7-8.
```

Example 7.7: Migration to OPTMODEL: Maximum Flow

The following example shows how to use PROC OPTMODEL to solve the example "Maximum Flow Problem" in Chapter 5, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The input data set is the same as in that example.

```
title 'Maximum Flow Problem';

data arcs;
   input _from_ $ _to_ $ _cost_ _capac_;
   datalines;
S a . .
S b . .
a c 1 7
b c 2 9
a d 3 5
b d 4 8
c e 5 15
d f 6 20
e g 7 11
```

```
f g 8 6
e h 9 12
f h 10 4
g T . .
h T . .
;
```

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called GOUT3:

```
proc optmodel;
   str source = 'S';
   str sink = 'T';

   set <str> NODES;
   num _supdem_ {i in NODES} = (if i in {source, sink} then . else 0);

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS} init 0;
   read data arcs nomiss into ARCS=[_from_ _to_] _cost_ _capac_;
   NODES = (union {<i,j> in ARCS} {i,j});

   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
   max obj = sum {<i,j> in ARCS: j = sink} Flow[i,j];
   con balance {i in NODES diff {source, sink}}:
      sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];

   solve;

   num _supply_ {<i,j> in ARCS} =
      (if _supdem_[i] ne 0 then _supdem_[i] else .);
   num _demand_ {<i,j> in ARCS} =
      (if _supdem_[j] ne 0 then -_supdem_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data gout3 from [_from_ _to_]
      _cost_ _capac_ _lo_ _supply_ _demand_ _flow_=Flow _fcost_;
quit;
```

To solve a maximum flow problem, you solve a network flow problem that has a zero supply or demand at all nodes other than the source and sink nodes, as specified in the declaration of the _SUPDEM_ numeric parameter and the balance constraint. The objective declaration uses the logical condition J = SINK to maximize the flow into the sink node. The output data set is displayed in Output 7.7.1.

Output 7.7.1 Optimal Solution

Obs	_from_	_to_	_cost_	_capac_	_lo_	_supply_	_demand_	_flow_	_fcost_
1	S	a	0	.	0	.	.	12	0
2	S	b	0	.	0	.	.	13	0
3	a	c	1	7	0	.	.	7	7
4	b	c	2	9	0	.	.	8	16
5	a	d	3	5	0	.	.	5	15
6	b	d	4	8	0	.	.	5	20
7	c	e	5	15	0	.	.	15	75
8	d	f	6	20	0	.	.	10	60
9	e	g	7	11	0	.	.	3	21
10	f	g	8	6	0	.	.	6	48
11	e	h	9	12	0	.	.	12	108
12	f	h	10	4	0	.	.	4	40
13	g	T	0	.	0	.	.	9	0
14	h	T	0	.	0	.	.	16	0

The log is displayed in Output 7.7.2.

Output 7.7.2 OPTMODEL Log

```
NOTE: There were 14 observations read from the data set WORK.ARCS.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 14 variables (0 free, 0 fixed).
NOTE: The problem has 8 linear constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 24 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The problem is a pure network instance. The ALGORITHM=NETWORK option is
      recommended for solving problems with this structure.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 10 variables and 6 constraints.
NOTE: The LP presolver removed 20 constraint coefficients.
NOTE: The presolved problem has 4 variables, 2 constraints, and 4 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                            Objective
      Phase Iteration         Value         Time
       D 1          1    0.000000E+00          0
       D 2          2    2.500000E+01          0
       P 2          5    2.500000E+01          0
NOTE: Optimal.
NOTE: Objective = 25.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.GOUT3 has 14 observations and 9 variables.
NOTE: The PROCEDURE OPTMODEL printed pages 10-11.
```

Example 7.8: Migration to OPTMODEL: Production, Inventory, Distribution

The following example shows how to use PROC OPTMODEL to solve the example "Production, Inventory, Distribution Problem" in Chapter 5, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The input data sets are the same as in that example.

```
title 'Minimum-Cost Flow Problem';
title2 'Production Planning/Inventory/Distribution';

data node0;
   input _node_ $ _supdem_ ;
   datalines;
fact1_1 1000
fact2_1 850
fact1_2 1000
fact2_2 1500
shop1_1 -900
shop2_1 -900
shop1_2 -900
shop2_2 -1450
;

data arc0;
   input _tail_ $ _head_ $ _cost_ _capac_ _lo_
      diagonal factory key_id $10. mth_made $ _name_&$17.;
   datalines;
fact1_1  f1_mar_1 127.9   500 50 19 1 production March prod f1 19 mar
fact1_1  f1_apr_1 78.6    600 50 19 1 production April prod f1 19 apl
fact1_1  f1_may_1 95.1    400 50 19 1 production May   .
f1_mar_1 f1_apr_1 15      50  .  19 1 storage    March .
f1_apr_1 f1_may_1 12      50  .  19 1 storage    April .
f1_apr_1 f1_mar_1 28      20  .  19 1 backorder  April back f1 19 apl
f1_may_1 f1_apr_1 28      20  .  19 1 backorder  May   back f1 19 may
f1_mar_1 f2_mar_1 11      .   .  19 . f1_to_2    March .
f1_apr_1 f2_apr_1 11      .   .  19 . f1_to_2    April .
f1_may_1 f2_may_1 16      .   .  19 . f1_to_2    May   .
f1_mar_1 shop1_1  -327.65 250 .  19 1 sales      March .
f1_apr_1 shop1_1  -300    250 .  19 1 sales      April .
f1_may_1 shop1_1  -285    250 .  19 1 sales      May   .
f1_mar_1 shop2_1  -362.74 250 .  19 1 sales      March .
f1_apr_1 shop2_1  -300    250 .  19 1 sales      April .
f1_may_1 shop2_1  -245    250 .  19 1 sales      May   .
fact2_1  f2_mar_1 88.0    450 35 19 2 production March prod f2 19 mar
fact2_1  f2_apr_1 62.4    480 35 19 2 production April prod f2 19 apl
fact2_1  f2_may_1 133.8   250 35 19 2 production May   .
f2_mar_1 f2_apr_1 18      30  .  19 2 storage    March .
f2_apr_1 f2_may_1 20      30  .  19 2 storage    April .
f2_apr_1 f2_mar_1 17      15  .  19 2 backorder  April back f2 19 apl
f2_may_1 f2_apr_1 25      15  .  19 2 backorder  May   back f2 19 may
f2_mar_1 f1_mar_1 10      40  .  19 . f2_to_1    March .
f2_apr_1 f1_apr_1 11      40  .  19 . f2_to_1    April .
f2_may_1 f1_may_1 13      40  .  19 . f2_to_1    May   .
```

```
f2_mar_1 shop1_1  -297.4  250 .  19 2 sales      March .
f2_apr_1 shop1_1  -290    250 .  19 2 sales      April .
f2_may_1 shop1_1  -292    250 .  19 2 sales      May   .
f2_mar_1 shop2_1  -272.7  250 .  19 2 sales      March .
f2_apr_1 shop2_1  -312    250 .  19 2 sales      April .
f2_may_1 shop2_1  -299    250 .  19 2 sales      May   .
fact1_2  f1_mar_2 217.9   400 40 25 1 production March prod f1 25 mar
fact1_2  f1_apr_2 174.5   550 50 25 1 production April prod f1 25 apl
fact1_2  f1_may_2 133.3   350 40 25 1 production May   .
f1_mar_2 f1_apr_2 20      40  .  25 1 storage    March .
f1_apr_2 f1_may_2 18      40  .  25 1 storage    April .
f1_apr_2 f1_mar_2 32      30  .  25 1 backorder  April back f1 25 apl
f1_may_2 f1_apr_2 41      15  .  25 1 backorder  May   back f1 25 may
f1_mar_2 f2_mar_2 23      .   .  25 . f1_to_2    March .
f1_apr_2 f2_apr_2 23      .   .  25 . f1_to_2    April .
f1_may_2 f2_may_2 26      .   .  25 . f1_to_2    May   .
f1_mar_2 shop1_2  -559.76 .   .  25 1 sales      March .
f1_apr_2 shop1_2  -524.28 .   .  25 1 sales      April .
f1_may_2 shop1_2  -475.02 .   .  25 1 sales      May   .
f1_mar_2 shop2_2  -623.89 .   .  25 1 sales      March .
f1_apr_2 shop2_2  -549.68 .   .  25 1 sales      April .
f1_may_2 shop2_2  -460.00 .   .  25 1 sales      May   .
fact2_2  f2_mar_2 182.0   650 35 25 2 production March prod f2 25 mar
fact2_2  f2_apr_2 196.7   680 35 25 2 production April prod f2 25 apl
fact2_2  f2_may_2 201.4   550 35 25 2 production May   .
f2_mar_2 f2_apr_2 28      50  .  25 2 storage    March .
f2_apr_2 f2_may_2 38      50  .  25 2 storage    April .
f2_apr_2 f2_mar_2 31      15  .  25 2 backorder  April back f2 25 apl
f2_may_2 f2_apr_2 54      15  .  25 2 backorder  May   back f2 25 may
f2_mar_2 f1_mar_2 20      25  .  25 . f2_to_1    March .
f2_apr_2 f1_apr_2 21      25  .  25 . f2_to_1    April .
f2_may_2 f1_may_2 43      25  .  25 . f2_to_1    May   .
f2_mar_2 shop1_2  -567.83 500 .  25 2 sales      March .
f2_apr_2 shop1_2  -542.19 500 .  25 2 sales      April .
f2_may_2 shop1_2  -461.56 500 .  25 2 sales      May   .
f2_mar_2 shop2_2  -542.83 500 .  25 2 sales      March .
f2_apr_2 shop2_2  -559.19 500 .  25 2 sales      April .
f2_may_2 shop2_2  -489.06 500 .  25 2 sales      May   .
;
```

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to SAS data sets called ARC1 and NODE2:

```
proc optmodel;
   set <str> NODES;
   num _supdem_ {NODES} init 0;
   read data node0 into NODES=[_node_] _supdem_;

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS};
   num diagonal {ARCS};
   num factory {ARCS};
```

```
   str key_id {ARCS};
   str mth_made {ARCS};
   str _name_ {ARCS};
   read data arc0 nomiss into ARCS=[_tail_ _head_] _lo_ _capac_ _cost_
      diagonal factory key_id mth_made _name_;
   NODES = NODES union (union {<i,j> in ARCS} {i,j});

   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
   min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
   con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];

   num infinity = constant('BIG');
   num excess = sum {i in NODES} _supdem_[i];
   if (excess > 0) then do;
      /* change equality constraint to le constraint for supply nodes */
      for {i in NODES: _supdem_[i] > 0} balance[i].lb = -infinity;
   end;
   else if (excess < 0) then do;
      /* change equality constraint to ge constraint for demand nodes */
      for {i in NODES: _supdem_[i] < 0} balance[i].ub = infinity;
   end;

   solve;

   num _supply_ {<i,j> in ARCS} =
      (if _supdem_[i] ne 0 then _supdem_[i] else .);
   num _demand_ {<i,j> in ARCS} =
      (if _supdem_[j] ne 0 then -_supdem_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data arc1 from [_tail_ _head_]
      _cost_ _capac_ _lo_ _name_ _supply_ _demand_ _flow_=Flow _fcost_
      _rcost_ =
         (if Flow[_tail_,_head_].rc ne 0 then Flow[_tail_,_head_].rc else .)
      _status_ = Flow.status diagonal factory key_id mth_made;
   create data node2 from [_node_]
      _supdem_ = (if _supdem_[_node_] ne 0 then _supdem_[_node_] else .)
      _dual_ = balance.dual;
quit;
```

The PROC OPTMODEL statements use both single-dimensional (NODES) and multiple-dimensional (ARCS) index sets, which are populated from the corresponding data set variables in the READ DATA statements. The _SUPDEM_, _LO_, and _CAPAC_ parameters are given initial values, and the NOMISS option in the READ DATA statement tells PROC OPTMODEL to read only the nonmissing values from the input data set. The balance constraint is initially declared as an equality, but depending on the total supply or demand, the sense of this constraint is changed to "≤" or "≥" by relaxing the constraint's lower or upper bound, respectively. The ARC1 output data set contains most of the same information as in the NETFLOW example, including reduced cost, basis status, and dual values. The _ANUMB_ and _TNUMB_ values do not apply here.

The PROC PRINT statements are similar to the PROC NETFLOW example:

```
options ls=80 ps=54;
proc print data=arc1 heading=h width=min;
   var _tail_ _head_ _cost_ _capac_ _lo_ _name_
   _supply_ _demand_ _flow_ _fcost_;
   sum _fcost_;
run;
proc print data=arc1 heading=h width=min;
   var _rcost_ _status_ diagonal factory key_id mth_made;
run;
proc print data=node2;
run;
```

The output data sets are displayed in Output 7.8.1.

Output 7.8.1 Output Data Sets

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_	_supply_	_demand_	_flow_	_fcost_
1	fact1_1	f1_mar_1	127.90	500	50	prod f1 19 mar	1000	.	345	44125.50
2	fact1_1	f1_apr_1	78.60	600	50	prod f1 19 apl	1000	.	600	47160.00
3	fact1_1	f1_may_1	95.10	400	50		1000	.	50	4755.00
4	f1_mar_1	f1_apr_1	15.00	50	0		.	.	0	0.00
5	f1_apr_1	f1_may_1	12.00	50	0		.	.	50	600.00
6	f1_apr_1	f1_mar_1	28.00	20	0	back f1 19 apl	.	.	20	560.00
7	f1_may_1	f1_apr_1	28.00	20	0	back f1 19 may	.	.	0	0.00
8	f1_mar_1	f2_mar_1	11.00	.	0		.	.	0	0.00
9	f1_apr_1	f2_apr_1	11.00	.	0		.	.	30	330.00
10	f1_may_1	f2_may_1	16.00	.	0		.	.	100	1600.00
11	f1_mar_1	shop1_1	-327.65	250	0		.	900	155	-50785.75
12	f1_apr_1	shop1_1	-300.00	250	0		.	900	250	-75000.00
13	f1_may_1	shop1_1	-285.00	250	0		.	900	0	0.00
14	f1_mar_1	shop2_1	-362.74	250	0		.	900	250	-90685.00
15	f1_apr_1	shop2_1	-300.00	250	0		.	900	250	-75000.00
16	f1_may_1	shop2_1	-245.00	250	0		.	900	0	0.00
17	fact2_1	f2_mar_1	88.00	450	35	prod f2 19 mar	850	.	290	25520.00
18	fact2_1	f2_apr_1	62.40	480	35	prod f2 19 apl	850	.	480	29952.00
19	fact2_1	f2_may_1	133.80	250	35		850	.	35	4683.00
20	f2_mar_1	f2_apr_1	18.00	30	0		.	.	0	0.00
21	f2_apr_1	f2_may_1	20.00	30	0		.	.	15	300.00
22	f2_apr_1	f2_mar_1	17.00	15	0	back f2 19 apl	.	.	0	0.00
23	f2_may_1	f2_apr_1	25.00	15	0	back f2 19 may	.	.	0	0.00
24	f2_mar_1	f1_mar_1	10.00	40	0		.	.	40	400.00
25	f2_apr_1	f1_apr_1	11.00	40	0		.	.	0	0.00
26	f2_may_1	f1_may_1	13.00	40	0		.	.	0	0.00
27	f2_mar_1	shop1_1	-297.40	250	0		.	900	250	-74350.00
28	f2_apr_1	shop1_1	-290.00	250	0		.	900	245	-71050.00
29	f2_may_1	shop1_1	-292.00	250	0		.	900	0	0.00
30	f2_mar_1	shop2_1	-272.70	250	0		.	900	0	0.00
31	f2_apr_1	shop2_1	-312.00	250	0		.	900	250	-78000.00
32	f2_may_1	shop2_1	-299.00	250	0		.	900	150	-44850.00
33	fact1_2	f1_mar_2	217.90	400	40	prod f1 25 mar	1000	.	400	87160.00
34	fact1_2	f1_apr_2	174.50	550	50	prod f1 25 apl	1000	.	550	95975.00
35	fact1_2	f1_may_2	133.30	350	40		1000	.	40	5332.00
36	f1_mar_2	f1_apr_2	20.00	40	0		.	.	0	0.00
37	f1_apr_2	f1_may_2	18.00	40	0		.	.	0	0.00
38	f1_apr_2	f1_mar_2	32.00	30	0	back f1 25 apl	.	.	30	960.00
39	f1_may_2	f1_apr_2	41.00	15	0	back f1 25 may	.	.	15	615.00
40	f1_mar_2	f2_mar_2	23.00	.	0		.	.	0	0.00
41	f1_apr_2	f2_apr_2	23.00	.	0		.	.	0	0.00
42	f1_may_2	f2_may_2	26.00	.	0		.	.	0	0.00
43	f1_mar_2	shop1_2	-559.76	.	0		.	900	0	0.00
44	f1_apr_2	shop1_2	-524.28	.	0		.	900	0	0.00
45	f1_may_2	shop1_2	-475.02	.	0		.	900	25	-11875.50
46	f1_mar_2	shop2_2	-623.89	.	0		.	1450	455	-283869.95
47	f1_apr_2	shop2_2	-549.68	.	0		.	1450	535	-294078.80
48	f1_may_2	shop2_2	-460.00	.	0		.	1450	0	0.00

Output 7.8.1 *continued*

Obs	_tail_	_head_	_cost_	_capac_	_lo_	_name_	_supply_	_demand_	_flow_	_fcost_
49	fact2_2	f2_mar_2	182.00	650	35	prod f2 25 mar	1500	.	645	117390.00
50	fact2_2	f2_apr_2	196.70	680	35	prod f2 25 apl	1500	.	680	133756.00
51	fact2_2	f2_may_2	201.40	550	35		1500	.	35	7049.00
52	f2_mar_2	f2_apr_2	28.00	50	0		.	.	0	0.00
53	f2_apr_2	f2_may_2	38.00	50	0		.	.	0	0.00
54	f2_apr_2	f2_mar_2	31.00	15	0	back f2 25 apl	.	.	0	0.00
55	f2_may_2	f2_apr_2	54.00	15	0	back f2 25 may	.	.	15	810.00
56	f2_mar_2	f1_mar_2	20.00	25	0		.	.	25	500.00
57	f2_apr_2	f1_apr_2	21.00	25	0		.	.	0	0.00
58	f2_may_2	f1_may_2	43.00	25	0		.	.	0	0.00
59	f2_mar_2	shop1_2	-567.83	500	0		.	900	500	-283915.00
60	f2_apr_2	shop1_2	-542.19	500	0		.	900	375	-203321.25
61	f2_may_2	shop1_2	-461.56	500	0		.	900	0	0.00
62	f2_mar_2	shop2_2	-542.83	500	0		.	1450	120	-65139.60
63	f2_apr_2	shop2_2	-559.19	500	0		.	1450	320	-178940.80
64	f2_may_2	shop2_2	-489.06	500	0		.	1450	20	-9781.20
										-1281110.35

Output 7.8.1 *continued*

Obs	_rcost_	_status_	diagonal	factory	key_id	mth_made
1	.	B	19	1	production	March
2	-0.65	U	19	1	production	April
3	0.85	L	19	1	production	May
4	63.65	L	19	1	storage	March
5	-3.00	U	19	1	storage	April
6	-20.65	U	19	1	backorder	April
7	43.00	L	19	1	backorder	May
8	50.90	L	19	.	f1_to_2	March
9	.	B	19	.	f1_to_2	April
10	.	B	19	.	f1_to_2	May
11	.	B	19	1	sales	March
12	-21.00	U	19	1	sales	April
13	9.00	L	19	1	sales	May
14	-46.09	U	19	1	sales	March
15	-32.00	U	19	1	sales	April
16	38.00	L	19	1	sales	May
17	.	B	19	2	production	March
18	-27.85	U	19	2	production	April
19	23.55	L	19	2	production	May
20	15.75	L	19	2	storage	March
21	.	B	19	2	storage	April
22	19.25	L	19	2	backorder	April
23	45.00	L	19	2	backorder	May
24	-29.90	U	19	.	f2_to_1	March
25	22.00	L	19	.	f2_to_1	April
26	29.00	L	19	.	f2_to_1	May
27	-9.65	U	19	2	sales	March
28	.	B	19	2	sales	April
29	18.00	L	19	2	sales	May
30	4.05	L	19	2	sales	March
31	-33.00	U	19	2	sales	April
32	.	B	19	2	sales	May
33	-45.16	U	25	1	production	March
34	-14.35	U	25	1	production	April
35	2.11	L	25	1	production	May
36	94.21	L	25	1	storage	March
37	75.66	L	25	1	storage	April
38	-42.21	U	25	1	backorder	April
39	-16.66	U	25	1	backorder	May
40	104.06	L	25	.	f1_to_2	March
41	13.49	L	25	.	f1_to_2	April
42	28.96	L	25	.	f1_to_2	May
43	47.13	L	25	1	sales	March
44	8.40	L	25	1	sales	April
45	.	B	25	1	sales	May
46	.	B	25	1	sales	March
47	.	B	25	1	sales	April
48	32.02	L	25	1	sales	May

Output 7.8.1 *continued*

Obs	_rcost_	_status_	diagonal	factory	key_id	mth_made
49	.	B	25	2	production	March
50	-1.66	U	25	2	production	April
51	73.17	L	25	2	production	May
52	11.64	L	25	2	storage	March
53	108.13	L	25	2	storage	April
54	47.36	L	25	2	backorder	April
55	-16.13	U	25	2	backorder	May
56	-61.06	U	25	.	f2_to_1	March
57	30.51	L	25	.	f2_to_1	April
58	40.04	L	25	.	f2_to_1	May
59	-42.00	U	25	2	sales	March
60	.	B	25	2	sales	April
61	10.50	L	25	2	sales	May
62	.	B	25	2	sales	March
63	.	B	25	2	sales	April
64	.	B	25	2	sales	May

Obs	_node_	_supdem_	_dual_
1	fact1_1	1000	0.00
2	fact2_1	850	0.00
3	fact1_2	1000	0.00
4	fact2_2	1500	0.00
5	shop1_1	-900	199.75
6	shop2_1	-900	188.75
7	shop1_2	-900	343.83
8	shop2_2	-1450	360.83
9	f1_mar_1	.	-127.90
10	f1_apr_1	.	-79.25
11	f1_may_1	.	-94.25
12	f2_mar_1	.	-88.00
13	f2_apr_1	.	-90.25
14	f2_may_1	.	-110.25
15	f1_mar_2	.	-263.06
16	f1_apr_2	.	-188.85
17	f1_may_2	.	-131.19
18	f2_mar_2	.	-182.00
19	f2_apr_2	.	-198.36
20	f2_may_2	.	-128.23

The log is displayed in Output 7.8.2.

Output 7.8.2 OPTMODEL Log

```
NOTE: There were 8 observations read from the data set WORK.NODE0.
NOTE: There were 64 observations read from the data set WORK.ARC0.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 64 variables (0 free, 0 fixed).
NOTE: The problem has 20 linear constraints (4 LE, 16 EQ, 0 GE, 0 range).
NOTE: The problem has 128 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 64 variables, 20 constraints, and 128
      constraint coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective
      Phase Iteration       Value         Time
       D 1          1    0.000000E+00         0
       D 2          2   -4.020320E+06         0
       D 2         32   -1.281110E+06         0
NOTE: Optimal.
NOTE: Objective = -1281110.35.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.ARC1 has 64 observations and 16 variables.
NOTE: The data set WORK.NODE2 has 20 observations and 3 variables.
NOTE: The PROCEDURE OPTMODEL printed pages 13-14.
```

Example 7.9: Migration to OPTMODEL: Shortest Path

The following example shows how to use PROC OPTMODEL to solve the example "Shortest Path Problem" in Chapter 5, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*). The input data set is the same as in that example.

```
title 'Shortest Path Problem';
title2 'How to get Hawaiian Pineapples to a London Restaurant';

data aircost1;
   input ffrom&$13. tto&$15. _cost_;
   datalines;
Honolulu       Chicago         105
Honolulu       San Francisco    75
Honolulu       Los Angeles      68
Chicago        Boston           45
Chicago        New York         56
San Francisco  Boston           71
San Francisco  New York         48
San Francisco  Atlanta          63
Los Angeles    New York         44
```

```
Los Angeles    Atlanta          57
Boston         Heathrow London  88
New York       Heathrow London  65
Atlanta        Heathrow London  76
;
```

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called SPATH:

```
proc optmodel;
   str sourcenode = 'Honolulu';
   str sinknode = 'Heathrow London';

   set <str> NODES;
   num _supdem_ {i in NODES} = (if i = sourcenode then 1
      else if i = sinknode then -1 else 0);

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS};
   read data aircost1 into ARCS=[ffrom tto] _cost_;
   NODES = (union {<i,j> in ARCS} {i,j});

   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
   con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j]
      - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];
   solve;

   num _supply_ {<i,j> in ARCS} =
      (if _supdem_[i] ne 0 then _supdem_[i] else .);
   num _demand_ {<i,j> in ARCS} =
      (if _supdem_[j] ne 0 then -_supdem_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data spath from [ffrom tto]
      _cost_ _capac_ _lo_ _supply_ _demand_ _flow_=Flow _fcost_
      _rcost_=(if Flow[ffrom,tto].rc ne 0 then Flow[ffrom,tto].rc else .)
      _status_=Flow.status;
quit;
```

The statements use both single-dimensional (NODES) and multiple-dimensional (ARCS) index sets. The ARCS index set is populated from the ffrom and tto data set variables in the READ DATA statement. To solve a shortest path problem, you solve a minimum-cost network flow problem that has a supply of one unit at the source node, a demand of one unit at the sink node, and zero supply or demand at all other nodes, as specified in the declaration of the _SUPDEM_ numeric parameter. The SPATH output data set contains most of the same information as in the PROC NETFLOW example, including reduced cost and basis status. The _ANUMB_ and _TNUMB_ values do not apply here.

The PROC PRINT statements are similar to the PROC NETFLOW example:

```
proc print data=spath;
   sum _fcost_;
run;
```

The output is displayed in Output 7.9.1.

Output 7.9.1 Output Data Set

Obs	ffrom	tto	_cost_	_capac_	_lo_	_supply_	_demand_	_flow_	_fcost_	_rcost_	_status_
1	Honolulu	Chicago	105	.	0	1	.	0	0	.	B
2	Honolulu	San Francisco	75	.	0	1	.	0	0	.	B
3	Honolulu	Los Angeles	68	.	0	1	.	1	68	.	B
4	Chicago	Boston	45	.	0	.	.	0	0	61	L
5	Chicago	New York	56	.	0	.	.	0	0	49	L
6	San Francisco	Boston	71	.	0	.	.	0	0	57	L
7	San Francisco	New York	48	.	0	.	.	0	0	11	L
8	San Francisco	Atlanta	63	.	0	.	.	0	0	37	L
9	Los Angeles	New York	44	.	0	.	.	1	44	.	B
10	Los Angeles	Atlanta	57	.	0	.	.	0	0	24	L
11	Boston	Heathrow London	88	.	0	.	1	0	0	.	B
12	New York	Heathrow London	65	.	0	.	1	1	65	.	B
13	Atlanta	Heathrow London	76	.	0	.	1	0	0	.	B
									177		

The log is displayed in Output 7.9.2.

Output 7.9.2 OPTMODEL Log

```
NOTE: There were 13 observations read from the data set WORK.AIRCOST1.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 13 variables (0 free, 0 fixed).
NOTE: The problem has 8 linear constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 26 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The problem is a pure network instance. The ALGORITHM=NETWORK option is
      recommended for solving problems with this structure.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed all variables and constraints.
NOTE: Optimal.
NOTE: Objective = 177.
NOTE: The data set WORK.SPATH has 13 observations and 11 variables.
NOTE: The PROCEDURE OPTMODEL printed pages 16-17.
```

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice-Hall.

Andersen, E. D., and Andersen, K. D. (1995). "Presolving in Linear Programming." *Mathematical Programming* 71:221–245.

Dantzig, G. B. (1963). *Linear Programming and Extensions*. Princeton, NJ: Princeton University Press.

Forrest, J. J., and Goldfarb, D. (1992). "Steepest-Edge Simplex Algorithms for Linear Programming." *Mathematical Programming* 5:1–28.

Gondzio, J. (1997). "Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method." *INFORMS Journal on Computing* 9:73–91.

Harris, P. M. J. (1973). "Pivot Selection Methods in the Devex LP Code." *Mathematical Programming* 57:341–374.

Maros, I. (2003). *Computational Techniques of the Simplex Method*. Boston: Kluwer Academic.

Chapter 8
The Mixed Integer Linear Programming Solver

Contents

Overview: MILP Solver **323**
Getting Started: MILP Solver **324**
Syntax: MILP Solver **325**
Functional Summary 325
MILP Solver Options 327
Details: MILP Solver **336**
Branch-and-Bound Algorithm 336
Controlling the Branch-and-Bound Algorithm 338
Presolve and Probing 340
Cutting Planes 340
Primal Heuristics 341
Parallel Processing 342
Node Log 342
Problem Statistics 343
Macro Variable _OROPTMODEL_ 344
Examples: MILP Solver **347**
Example 8.1: Scheduling 347
Example 8.2: Multicommodity Transshipment Problem with Fixed Charges 351
Example 8.3: Facility Location 357
Example 8.4: Traveling Salesman Problem 368
References **374**

Overview: MILP Solver

The OPTMODEL procedure provides a framework for specifying and solving mixed integer linear programs (MILPs). A standard mixed integer linear program has the formulation

$$\begin{array}{rl} \min & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \\ & \mathbf{x}_i \in \mathbb{Z} \quad \forall i \in \mathcal{S} \end{array} \qquad \text{(MILP)}$$

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of structural variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the matrix of technological coefficients
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^n$	is the vector of lower bounds on variables
$\mathbf{u}$	$\in$	$\mathbb{R}^n$	is the vector of upper bounds on variables
$\mathcal{S}$			is a nonempty subset of the set $\{1 \ldots, n\}$ of indices

The MILP solver, available in the OPTMODEL procedure, implements a linear-programming-based branch-and-cut algorithm. This divide-and-conquer approach attempts to solve the original problem by solving linear programming relaxations of a sequence of smaller subproblems. The MILP solver also implements advanced techniques such as presolving, generating cutting planes, and applying primal heuristics to improve the efficiency of the overall algorithm.

The MILP solver provides various control options and solution strategies. In particular, you can enable, disable, or set levels for the advanced techniques previously mentioned. It is also possible to input an incumbent solution; see the section "Warm Start Option" on page 327 for details.

Getting Started: MILP Solver

The following example illustrates how you can use the OPTMODEL procedure to solve mixed integer linear programs. For more examples, see the section "Examples: MILP Solver" on page 347. Suppose you want to solve the following problem:

$$
\begin{array}{lrcrcrcrl}
\min & 2x_1 & - & 3x_2 & - & 4x_3 & & & \\
\text{s.t.} & & - & 2x_2 & - & 3x_3 & \geq & -5 & \text{(R1)} \\
& x_1 & + & x_2 & + & 2x_3 & \leq & 4 & \text{(R2)} \\
& x_1 & + & 2x_2 & + & 3x_3 & \leq & 7 & \text{(R3)} \\
& & & x_1, & x_2, & x_3 & \geq & 0 & \\
& & & x_1, & x_2, & x_3 & \in \mathbb{Z} & &
\end{array}
$$

You can use the following statements to call the OPTMODEL procedure for solving mixed integer linear programs:

```
proc optmodel;
   var x{1..3} >= 0 integer;

   min f = 2*x[1] - 3*x[2] - 4*x[3];

   con r1: -2*x[2] - 3*x[3] >= -5;
   con r2: x[1] + x[2] + 2*x[3] <= 4;
   con r3: x[1] + 2*x[2] + 3*x[3] <= 7;

   solve with milp / presolver = automatic heuristics = automatic;
   print x;
quit;
```

The PRESOLVER= and HEURISTICS= options specify the levels for presolving and applying heuristics, respectively. In this example, each option is set to its default value, AUTOMATIC, meaning that the solver automatically determines the appropriate levels for presolve and heuristics.

The optimal value of x is shown in Figure 8.1.

Figure 8.1 Solution Output

The OPTMODEL Procedure

[1]	x
1	0
2	1
3	1

The solution summary stored in the macro variable _OROPTMODEL_ can be viewed by issuing the following statement:

```
%put &_OROPTMODEL_;
```

This statement produces the output shown in Figure 8.2.

Figure 8.2 Macro Output

```
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=-7 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0 BEST_BOUND=-7 NODES=1 ITERATIONS=4 PRESOLVE_TIME=0.00
SOLUTION_TIME=0.00
```

Syntax: MILP Solver

The following statement is available in the OPTMODEL procedure:

SOLVE WITH MILP < / *options* > **;**

Functional Summary

Table 8.1 summarizes the options available for the SOLVE WITH MILP statement, classified by function.

Table 8.1 Options for the MILP Solver

Description	Option
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Warm Start Option	
Specifies the input primal solution (warm start)	PRIMALIN
Control Options	
Specifies the stopping criterion based on absolute objective gap	ABSOBJGAP=
Specifies the cutoff value for node removal	CUTOFF=

Table 8.1 (continued)

Description	Option
Emphasizes feasibility or optimality	EMPHASIS=
Specifies the maximum violation on variables and constraints	FEASTOL=
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=
Specifies the frequency of printing the node log	LOGFREQ=
Specifies the detail of solution progress printed in log	LOGLEVEL=
Specifies the maximum number of nodes to be processed	MAXNODES=
Specifies the maximum number of solutions to be found	MAXSOLS=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the tolerance used in determining the optimality of nodes in the branch-and-bound tree	OPTTOL=
Specifies the probing level	PROBE=
Specifies the stopping criterion based on relative objective gap	RELOBJGAP=
Specifies the scale of the problem matrix	SCALE=
Specifies the initial seed for the random number generator	SEED=
Specifies the stopping criterion based on target objective value	TARGET=
Specifies whether time units are CPU time or real time	TIMETYPE=
Heuristics Option	
Specifies the primal heuristics level	HEURISTICS=
Search Options	
Specifies the level of conflict search	CONFLICTSEARCH=
Specifies the node selection strategy	NODESEL=
Enables use of variable priorities	PRIORITY=
Specifies the restarting strategy	RESTARTS=
Specifies the number of simplex iterations performed on each variable in strong branching strategy	STRONGITER=
Specifies the number of candidates for strong branching	STRONGLEN=
Specifies the level of symmetry detection	SYMMETRY=
Specifies the rule for selecting branching variable	VARSEL=
Cut Options	
Specifies the cut level for all cuts	ALLCUTS=
Specifies the clique cut level	CUTCLIQUE=
Specifies the flow cover cut level	CUTFLOWCOVER=
Specifies the flow path cut level	CUTFLOWPATH=
Specifies the Gomory cut level	CUTGOMORY=
Specifies the generalized upper bound (GUB) cover cut level	CUTGUB=
Specifies the implied bounds cut level	CUTIMPLIED=
Specifies the knapsack cover cut level	CUTKNAPSACK=
Specifies the lift-and-project cut level	CUTLAP=
Specifies the mixed lifted 0-1 cut level	CUTMILIFTED=
Specifies the mixed integer rounding (MIR) cut level	CUTMIR=
Specifies the multicommodity network flow cut level	CUTMULTICOMMODITY=
Specifies the row multiplier factor for cuts	CUTSFACTOR=
Specifies the overall cut aggressiveness	CUTSTRATEGY=
Specifies the zero-half cut level	CUTZEROHALF=

Table 8.1 (continued)

Description	Option
Decomposition Algorithm Options	
Enables decomposition algorithm and specifies general control options	DECOMP=()
Specifies options for the master problem	DECOMP_MASTER=()
Specifies options for the master problem solved as a MILP	DECOMP_MASTER_IP=()
Specifies options for the subproblem	DECOMP_SUBPROB=()

MILP Solver Options

This section describes the options that are recognized by the MILP solver in PROC OPTMODEL. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the MILP solver is explicitly specified using a WITH clause. For example, the following line could appear in PROC OPTMODEL statements:

```
solve with milp / allcuts=aggressive maxnodes=10000 primalin;
```

Presolve Option

PRESOLVER=*number* | *string*

specifies a presolve *string* or its corresponding value *number*, as listed in Table 8.2.

Table 8.2 Values for PRESOLVER= Option

number	*string*	**Description**
–1	AUTOMATIC	Applies the default level of presolve processing
0	NONE	Disables presolver
1	BASIC	Performs minimal presolve processing
2	MODERATE	Applies a higher level of presolve processing
3	AGGRESSIVE	Applies the highest level of presolve processing

The default value is AUTOMATIC.

Warm Start Option

PRIMALIN

enables you to input a starting solution in PROC OPTMODEL before invoking the MILP solver. Adding the PRIMALIN option to the SOLVE statement requests that the MILP solver use the current variable values as a starting solution (warm start). If the MILP solver finds that the input solution is feasible, then the input solution provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution is not feasible, the MILP solver tries to repair it. It is possible to set a variable value to the missing value '.' to mark a variable for repair. When it is difficult to find a good integer feasible solution for a problem, warm start can reduce solution time significantly.

NOTE: If the MILP solver produces a feasible solution, the variable values from that run can be used as the warm start solution for a subsequent run. If the warm start solution is not feasible for the subsequent run, the solver automatically tries to repair it.

Control Options

ABSOBJGAP=*number*

specifies a stopping criterion. When the absolute difference between the best integer objective and the best bound on the objective function value falls below the value of *number*, the MILP solver stops. The value of *number* can be any nonnegative number; the default value is 1E–6.

CUTOFF=*number*

cuts off any nodes in a minimization (maximization) problem that have an objective value at or above (below) *number*. The value of *number* can be any number; the default value is the positive (negative) number that has the largest absolute value representable in your operating environment.

EMPHASIS=*number* | *string*

specifies a search emphasis *string* or its corresponding value *number* as listed in Table 8.3.

Table 8.3 Values for EMPHASIS= Option

number	*string*	**Description**
0	BALANCE	Performs a balanced search
1	OPTIMAL	Emphasizes optimality over feasibility
2	FEASIBLE	Emphasizes feasibility over optimality

The default value is BALANCE.

FEASTOL=*number*

specifies the tolerance that the MILP solver uses to check the feasibility of a solution. This tolerance applies both to the maximum violation of bounds on variables and to the difference between the right-hand sides and left-hand sides of constraints. The value of *number* can be any value between 1E–4 and 1E–9, inclusive. However, the value of *number* cannot be larger than the integer feasibility tolerance. If the value of *number* is larger than the value of the INTTOL= option, then the solver sets FEASTOL= to the value of INTTOL=. The default value is 1E–6.

If the MILP solver fails to find a feasible solution within this tolerance but does find a solution that has some violation, then the solver stops with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMODEL_ " on page 344).

INTTOL=*number*

specifies the amount by which an integer variable value can differ from an integer and still be considered integer feasible. The value of *number* can be any number between 1E–9 and 0.5, inclusive. The MILP solver attempts to find an optimal solution whose integer infeasibility is less than *number*. The default value is 1E–5.

If the best solution that the solver finds has an integer infeasibility larger than the value of *number*, then the solver stops with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMODEL_ " on page 344).

LOGFREQ=*number*

PRINTFREQ=*number*

specifies how often information is printed in the node log. The value of *number* can be any nonnegative number up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of *number* is 100. If *number* is set to 0, then the node log is disabled. If *number* is positive, then an entry is made in the node log at the first node, at the last node, and at intervals dictated by the value of *number*. An entry is also made each time a better integer solution is found.

LOGLEVEL=*number* | *string*

PRINTLEVEL2=*number* | *string*

controls the amount of information displayed in the SAS log by the MILP solver, from a short description of presolve information and summary to details at each node. Table 8.4 describes the valid values for this option.

Table 8.4 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all solver-related messages to SAS log
1	BASIC	Displays a solver summary after stopping
2	MODERATE	Prints a solver summary and a node log by using the interval dictated by the LOGFREQ= option
3	AGGRESSIVE	Prints a detailed solver summary and a node log by using the interval dictated by the LOGFREQ= option

The default value is MODERATE.

MAXNODES=*number*

specifies the maximum number of branch-and-bound nodes to be processed. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of *number* is $2^{31} - 1$.

MAXSOLS=*number*

specifies a stopping criterion. If *number* solutions have been found, then the solver stops. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of *number* is $2^{31} - 1$.

MAXTIME=*t*

specifies an upper limit of *t* units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OPTTOL=*number*

specifies the tolerance used to determine the optimality of nodes in the branch-and-bound tree. The value of *number* can be any value between (and including) 1E–4 and 1E–9. The default is 1E–6.

PROBE=*number* | *string*
specifies a probing *string* or its corresponding value *number*, as listed in Table 8.5.

Table 8.5 Values for PROBE= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses the probing strategy determined by the MILP solver
0	NONE	Disables probing
1	MODERATE	Uses probing moderately
2	AGGRESSIVE	Uses probing aggressively

The default value is AUTOMATIC.

RELOBJGAP=*number*
specifies a stopping criterion based on the best integer objective (BestInteger) and the best bound on the objective function value (BestBound). The relative objective gap is equal to

$$| \text{BestInteger} - \text{BestBound} | / (1\text{E}{-}10 + | \text{BestBound} |)$$

When this value becomes smaller than the specified gap size *number*, the MILP solver stops. The value of *number* can be any nonnegative number; the default value is 1E–4.

SCALE=*option*
indicates whether to scale the problem matrix. SCALE= can take either of the values AUTOMATIC (–1) and NONE (0). SCALE=AUTOMATIC scales the matrix as determined by the MILP solver; SCALE=NONE disables scaling. The default value is AUTOMATIC.

SEED=*number*
specifies the initial seed of the random number generator. This option affects the perturbation in the simplex solvers; thus it might result in a different optimal solution and a different solver path. This option usually has a significant, but unpredictable, effect on the solution time. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of the seed is 100.

TARGET=*number*
specifies a stopping criterion for minimization (maximization) problems. If the best integer objective is better than or equal to *number*, the solver stops. The value of *number* can be any number; the default value is the negative (positive) number that has the largest absolute value representable in your operating environment.

TIMETYPE=*string* | *number*
specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. Table 8.6 describes the valid values of the TIMETYPE= option.

Table 8.6 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

The "Optimization Statistics" table, an output of PROC OPTMODEL if you specify PRINTLEVEL=2 in the PROC OPTMODEL statement, also includes the same time units for Presolver Time and Solver Time. The other times (such as Problem Generation Time) in the "Optimization Statistics" table are also in the same units.

The default value of the TIMETYPE= option depends on the algorithm used and on various options. When the solver is used with distributed or multithreaded processing, then by default TIMETYPE= REAL. Otherwise, by default TIMETYPE= CPU. Table 8.7 describes the detailed logic for determining the default; the first context in the table that applies determines the default value. The NTHREADS= and NODES= options are specified in the PERFORMANCE statement of the OPTMODEL procedure. For more information about the NTHREADS= and NODES= options, see the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Table 8.7 Default Value for TIMETYPE= Option

Context	Default
Solver is invoked in an OPTMODEL COFOR loop	REAL
NODES= value is nonzero for the decomposition algorithm	REAL
NTHREADS= value is greater than 1 and NODES=0 for the decomposition algorithm	REAL
NTHREADS= value is greater than 1	REAL
Otherwise	CPU

Heuristics Option

HEURISTICS=*number* | *string*

controls the level of primal heuristics applied by the MILP solver. This level determines how frequently primal heuristics are applied during the branch-and-bound tree search. It also affects the maximum number of iterations allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. The values of *string* and the corresponding values of *number* are listed in Table 8.8.

Table 8.8 Values for HEURISTICS= Option

number	*string*	Description
–1	AUTOMATIC	Applies default level of heuristics, similar to MODERATE
0	NONE	Disables all primal heuristics
1	BASIC	Applies basic primal heuristics at low frequency
2	MODERATE	Applies most primal heuristics at moderate frequency
3	AGGRESSIVE	Applies all primal heuristics at high frequency

Setting HEURISTICS=NONE does not disable the heuristics that repair an infeasible input solution that is specified by using the PRIMALIN option.

The default value is AUTOMATIC. For details about primal heuristics, see the section "Primal Heuristics" on page 341.

Search Options

CONFLICTSEARCH=*number* | *string*

specifies the level of conflict search performed by the MILP solver. Conflict finds clauses resulting from infeasible subproblems that arise in the search tree. The values of *string* and the corresponding values of *number* are listed in Table 8.9.

Table 8.9 Values for CONFLICTSEARCH= Option

number	*string*	**Description**
–1	AUTOMATIC	Performs conflict search based on a strategy determined by the MILP solver
0	NONE	Disables conflict search
1	MODERATE	Performs a moderate conflict search
2	AGGRESSIVE	Performs an aggressive conflict search

The default value is AUTOMATIC.

NODESEL=*number* | *string*

specifies the node selection strategy *string* or its corresponding value *number* as listed in Table 8.10.

Table 8.10 Values for NODESEL= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses automatic node selection
0	BESTBOUND	Chooses the node with the best relaxed objective (best-bound-first strategy)
1	BESTESTIMATE	Chooses the node with the best estimate of the integer objective value (best-estimate-first strategy)
2	DEPTH	Chooses the most recently created node (depth-first strategy)

The default value is AUTOMATIC. For details about node selection, see the section "Node Selection" on page 338.

PRIORITY=0 | **1**

indicates whether to use specified branching priorities for integer variables. PRIORITY=0 ignores variable priorities; PRIORITY=1 uses priorities when they exist. The default value is 1. See the section "Branching Priorities" on page 339 for details.

RESTARTS=*number* | *string*

specifies the strategy for restarting the processing of the root node. The values of *string* and the corresponding values of *number* are listed in Table 8.11.

Table 8.11 Values for RESTARTS= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses a restarting strategy determined by the MILP solver

Table 8.11 (continued)

number	*string*	**Description**
0	NONE	Disables restarting
1	BASIC	Uses a basic restarting strategy
2	MODERATE	Uses a moderate restarting strategy
3	AGGRESSIVE	Uses an aggressive restarting strategy

The default value is AUTOMATIC.

STRONGITER=*number* | **AUTOMATIC**

specifies the number of simplex iterations performed for each variable in the candidate list when the strong branching variable selection strategy is used. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, the MILP solver uses the default value; this value is calculated automatically.

STRONGLEN=*number* | **AUTOMATIC**

specifies the number of candidates used when the strong branching variable selection strategy is performed. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, the MILP solver uses the default value; this value is calculated automatically.

SYMMETRY=*number* | *string*

specifies the level of symmetry detection. Symmetry detection identifies groups of equivalent decision variables and uses this information to solve the problem more efficiently. The values of *string* and the corresponding values of *number* are listed in Table 8.12.

Table 8.12 Values for SYMMETRY= Option

number	*string*	**Description**
–1	AUTOMATIC	Performs symmetry detection based on a strategy that is determined by the MILP solver
0	NONE	Disables symmetry detection
1	BASIC	Performs a basic symmetry detection
2	MODERATE	Performs a moderate symmetry detection
3	AGGRESSIVE	Performs an aggressive symmetry detection

The default value is AUTOMATIC. For more information about symmetry detection, see (Ostrowski 2008).

VARSEL=*number* | *string*

specifies the rule for selecting the branching variable. The values of *string* and the corresponding values of *number* are listed in Table 8.13.

Table 8.13 Values for VARSEL= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses automatic branching variable selection
0	MAXINFEAS	Chooses the variable with maximum infeasibility

Table 8.13 (continued)

number	*string*	**Description**
1	MININFEAS	Chooses the variable with minimum infeasibility
2	PSEUDO	Chooses a branching variable based on pseudocost
3	STRONG	Uses strong branching variable selection strategy

The default value is AUTOMATIC. For details about variable selection, see the section "Variable Selection" on page 339.

Cut Options

Table 8.14 describes the *string* and *number* values for the cut options in the OPTMODEL procedure.

Table 8.14 Values for Individual Cut Options

number	*string*	**Description**
–1	AUTOMATIC	Generates cutting planes based on a strategy determined by the MILP solver
0	NONE	Disables generation of cutting planes
1	MODERATE	Uses a moderate cut strategy
2	AGGRESSIVE	Uses an aggressive cut strategy

You can specify the CUTSTRATEGY= option to set the overall aggressiveness of the cut generation in the MILP solver. Alternatively, you can use the ALLCUTS= option to set all cut types to the same level. You can override the ALLCUTS= value by using the options that correspond to particular cut types. For example, if you want the MILP solver to generate only Gomory cuts, specify ALLCUTS=NONE and CUTGOMORY=AUTOMATIC. If you want to generate all cuts aggressively but generate no lift-and-project cuts, set ALLCUTS=AGGRESSIVE and CUTLAP=NONE.

ALLCUTS=*number* | *string*
: provides a shorthand way of setting all the cuts-related options in one setting. In other words, ALLCUTS=*number* is equivalent to setting each of the individual cuts parameters to the same value *number*. Thus, ALLCUTS=–1 has the effect of setting CUTCLIQUE=–1, CUTFLOWCOVER=–1, CUTFLOWPATH=–1, . . . , CUTMULTICOMMODITY=–1, and CUTZEROHALF=–1. Table 8.14 lists the values that can be assigned to *option* and *number*. In addition, you can override levels for individual cuts with the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIMPLIED=, CUTKNAPSACK=, CUTLAP=, CUTMILIFTED=, CUTMIR=, CUTMULTICOMMODITY=, and CUTZEROHALF= options. If the ALLCUTS= option is not specified, then all the cuts-related options are either at their individually specified values (if the corresponding option is specified) or at their default values (if that option is not specified).

CUTCLIQUE=*number* | *string*
: specifies the level of clique cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTCLIQUE= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTFLOWCOVER=*number* | *string*
specifies the level of flow cover cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTFLOWCOVER= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTFLOWPATH=*number* | *string*
specifies the level of flow path cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTFLOWPATH= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTGOMORY=*number* | *string*
specifies the level of Gomory cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTGOMORY= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTGUB=*number* | *string*
specifies the level of generalized upper bound (GUB) cover cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTGUB= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTIMPLIED=*number* | *string*
specifies the level of implied bound cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTIMPLIED= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTKNAPSACK=*number* | *string*
specifies the level of knapsack cover cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTKNAPSACK= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTLAP=*number* | *string*
specifies the level of lift-and-project (LAP) cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTLAP= option overrides the ALLCUTS= option. The default value is NONE.

CUTMILIFTED=*number* | *string*
specifies the level of mixed lifted 0-1 cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTMILIFTED= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTMIR=*number* | *string*
specifies the level of mixed integer rounding (MIR) cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTMIR= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTMULTICOMMODITY=*number* | *string*
specifies the level of multicommodity network flow cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*. The CUTMULTICOMMODITY= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTSFACTOR=*number*
specifies a row multiplier factor for cuts. The number of cuts that are added is limited to *number* times the original number of rows. The value of *number* can be any nonnegative number less than or equal to 100; the default value is automatically calculated by the MILP solver.

CUTSTRATEGY=*number* | *string*

CUTS=*number* | *string*
specifies the overall aggressiveness of the cut generation in the solver. Setting a nondefault value adjusts a number of cut parameters such that the cut generation is basic, moderate, or aggressive compared to the default value.

CUTZEROHALF=*number* | *string*
specifies the level of zero-half cuts that are generated by the MILP solver. Table 8.14 lists the values that can be assigned to *option* and *number*.The CUTZEROHALF= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

Decomposition Algorithm Options

The following options are available for the decomposition algorithm in the MILP solver. For information about the decomposition algorithm, see Chapter 15, "The Decomposition Algorithm."

DECOMP=(*options***)**
enables the decomposition algorithm and specifies overall control options for the algorithm. For more information about this option, see Chapter 15, "The Decomposition Algorithm."

DECOMP_MASTER=(*options***)**
specifies options for the master problem. For more information about this option, see Chapter 15, "The Decomposition Algorithm."

DECOMP_MASTER_IP=(*options***)**
specifies options for the (restricted) master problem solved as a MILP with the current set of columns in an effort to obtain an integer feasible solution. For more information about this option, see Chapter 15, "The Decomposition Algorithm."

DECOMP_SUBPROB=(*options***)**
specifies option for the subproblem. For more information about this option, see Chapter 15, "The Decomposition Algorithm."

Details: MILP Solver

Branch-and-Bound Algorithm

The branch-and-bound algorithm, first proposed by Land and Doig (1960), is an effective approach to solving mixed integer linear programs. The following discussion outlines the approach and explains how to enhance its progress by using several advanced techniques.

The branch-and-bound algorithm solves a mixed integer linear program by dividing the search space and generating a sequence of subproblems. The search space of a mixed integer linear program can be represented by a tree. Each node in the tree is identified with a subproblem derived from previous subproblems on the path that leads to the root of the tree. The subproblem (MILP^0) associated with the root is identical to the original problem, which is called (MILP), given in the section "Overview: MILP Solver" on page 323.

The linear programming relaxation (LP^0) of (MILP^0) can be written as

$$\begin{aligned} \min \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

The branch-and-bound algorithm generates subproblems along the nodes of the tree by using the following scheme. Consider $\bar{x}^0$, the optimal solution to (LP^0), which is usually obtained by using the dual simplex algorithm. If $\bar{x}_i^0$ is an integer for all $i \in \mathcal{S}$, then $\bar{x}^0$ is an optimal solution to (MILP). Suppose that for some $i \in \mathcal{S}$, $\bar{x}_i^0$ is nonintegral. In that case the algorithm defines two new subproblems (MILP^1) and (MILP^2), descendants of the parent subproblem (MILP^0). The subproblem (MILP^1) is identical to (MILP^0) except for the additional constraint

$$x_i \leq \lfloor \bar{x}_i^0 \rfloor$$

and the subproblem (MILP^2) is identical to (MILP^0) except for the additional constraint

$$x_i \geq \lceil \bar{x}_i^0 \rceil$$

The notation $\lfloor y \rfloor$ represents the largest integer that is less than or equal to y, and the notation $\lceil y \rceil$ represents the smallest integer that is greater than or equal to y. The two preceding constraints can be handled by modifying the bounds of the variable x_i rather than by explicitly adding the constraints to the constraint matrix. The two new subproblems do not have $\bar{x}^0$ as a feasible solution, but the integer solution to (MILP) must satisfy one of the preceding constraints. The two subproblems thus defined are called *active nodes* in the branch-and-bound tree, and the variable x_i is called the *branching variable*.

In the next step the branch-and-bound algorithm chooses one of the active nodes and attempts to solve the linear programming relaxation of that subproblem. The relaxation might be infeasible, in which case the subproblem is dropped (fathomed). If the subproblem can be solved and the solution is *integer feasible* (that is, x_i is an integer for all $i \in \mathcal{S}$), then its objective value provides an *upper bound* for the objective value in the minimization problem (MILP); if the solution is not integer feasible, then it defines two new subproblems. Branching continues in this manner until there are no active nodes. At this point the best integer solution found is an optimal solution for (MILP). If no integer solution has been found, then (MILP) is integer infeasible. You can specify other criteria to stop the branch-and-bound algorithm before it processes all the active nodes; see the section "Controlling the Branch-and-Bound Algorithm" on page 338 for details.

Upper bounds from integer feasible solutions can be used to *fathom* or *cut off* active nodes. Since the objective value of an optimal solution cannot be greater than an upper bound, active nodes with lower bounds higher than an existing upper bound can be safely deleted. In particular, if z is the objective value of the current best integer solution, then any active subproblems whose relaxed objective value is greater than or equal to z can be discarded.

It is important to realize that mixed integer linear programs are non-deterministic polynomial-time hard (NP-hard). Roughly speaking, this means that the effort required to solve a mixed integer linear program

grows exponentially with the size of the problem. For example, a problem with 10 binary variables can generate in the worst case $2^{10} = 1,024$ nodes in the branch-and-bound tree. A problem with 20 binary variables can generate in the worst case $2^{20} = 1,048,576$ nodes in the branch-and-bound tree. Although it is unlikely that the branch-and-bound algorithm has to generate every single possible node, the need to explore even a small fraction of the potential number of nodes for a large problem can be resource-intensive.

A number of techniques can speed up the search progress of the branch-and-bound algorithm. Heuristics are used to find feasible solutions, which can improve the upper bounds on solutions of mixed integer linear programs. Cutting planes can reduce the search space and thus improve the lower bounds on solutions of mixed integer linear programs. When using cutting planes, the branch-and-bound algorithm is also called the *branch-and-cut algorithm*. Preprocessing can reduce problem size and improve problem solvability. The MILP solver in PROC OPTMODEL employs various heuristics, cutting planes, preprocessing, and other techniques, which you can control through corresponding options.

Controlling the Branch-and-Bound Algorithm

There are numerous strategies that can be used to control the branch-and-bound search (see Linderoth and Savelsbergh 1998, Achterberg, Koch, and Martin 2005). The MILP solver in PROC OPTMODEL implements the most widely used strategies and provides several options that enable you to direct the choice of the next active node and of the branching variable. In the discussion that follows, let (LP^k) be the linear programming relaxation of subproblem (MILP^k). Also, let

$$f_i(k) = \bar{x}_i^k - \lfloor \bar{x}_i^k \rfloor$$

where $\bar{x}^k$ is the optimal solution to the relaxation problem (LP^k) solved at node k.

Node Selection

The NODESEL= option specifies the strategy used to select the next active node. The valid keywords for this option are AUTOMATIC, BESTBOUND, BESTESTIMATE, and DEPTH. The following list describes the strategy associated with each keyword:

AUTOMATIC — enables the MILP solver to choose the best node selection strategy based on problem characteristics and search progress. This is the default setting.

BESTBOUND — chooses the node with the smallest (or largest, in the case of a maximization problem) relaxed objective value. The best-bound strategy tends to reduce the number of nodes to be processed and can improve lower bounds quickly. However, if there is no good upper bound, the number of active nodes can be large. This can result in the solver running out of memory.

BESTESTIMATE — chooses the node with the smallest (or largest, in the case of a maximization problem) objective value of the estimated integer solution. Besides improving lower bounds, the best-estimate strategy also attempts to process nodes that can yield good feasible solutions.

DEPTH — chooses the node that is deepest in the search tree. Depth-first search is effective in locating feasible solutions, since such solutions are usually deep in the search tree. Compared to the costs of the best-bound and best-estimate strategies, the cost of solving LP relaxations is less in the depth-first strategy. The number of active nodes is

generally small, but it is possible that the depth-first search will remain in a portion of the search tree with no good integer solutions. This occurrence is computationally expensive.

Variable Selection

The VARSEL= option specifies the strategy used to select the next branching variable. The valid keywords for this option are AUTOMATIC, MAXINFEAS, MININFEAS, PSEUDO, and STRONG. The following list describes the action taken in each case when $\bar{x}^k$, a relaxed optimal solution of (MILP^k), is used to define two active subproblems. In the following list, "INTTOL" refers to the value assigned using the INTTOL= option. For details about the INTTOL= option, see the section "Control Options" on page 328.

AUTOMATIC enables the MILP solver to choose the best variable selection strategy based on problem characteristics and search progress. This is the default setting.

MAXINFEAS chooses as the branching variable the variable x_i such that i maximizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and}$$

$$\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

MININFEAS chooses as the branching variable the variable x_i such that i minimizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and}$$

$$\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

PSEUDO chooses as the branching variable the variable x_i such that i maximizes the weighted up and down pseudocosts. Pseudocost branching attempts to branch on significant variables first, quickly improving lower bounds. Pseudocost branching estimates significance based on historical information; however, this approach might not be accurate for future search.

STRONG chooses as the branching variable the variable x_i such that i maximizes the estimated improvement in the objective value. Strong branching first generates a list of candidates, then branches on each candidate and records the improvement in the objective value. The candidate with the largest improvement is chosen as the branching variable. Strong branching can be effective for combinatorial problems, but it is usually computationally expensive.

Branching Priorities

In some cases, it is possible to speed up the branch-and-bound algorithm by branching on variables in a specific order. You can accomplish this in PROC OPTMODEL by attaching branching priorities to the integer variables in your model by using the .priority suffix. More information about this suffix is available in the section "Integer Variable Suffixes" on page 137 in Chapter 5. For an example in which branching priorities are used, see Example 8.3.

Presolve and Probing

The MILP solver in PROC OPTMODEL includes a variety of presolve techniques to reduce problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix. Presolve might also modify the constraint coefficients to tighten the formulation of the problem.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels by specifying the PRESOLVER= option.

Probing is a technique that tentatively sets each binary variable to 0 or 1, then explores the logical consequences (Savelsbergh 1994). Probing can expedite the solution of a difficult problem by fixing variables and improving the model. However, probing is often computationally expensive and can significantly increase the solution time in some cases. You can enable probing at different levels by specifying the PROBE= option.

Cutting Planes

The feasible region of every linear program forms a *polyhedron*. Every polyhedron in n-space can be written as a finite number of half-spaces (equivalently, inequalities). In the notation used in this chapter, this polyhedron is defined by the set $\mathcal{Q} = \{x \in \mathbb{R}^n \mid Ax \leq b, l \leq x \leq u\}$. After you add the restriction that some variables must be integral, the set of feasible solutions, $\mathcal{F} = \{x \in \mathcal{Q} \mid x_i \in \mathbb{Z} \ \forall i \in \mathcal{S}\}$, no longer forms a polyhedron.

The *convex hull* of a set X is the minimal convex set that contains X. In solving a mixed integer linear program, in order to take advantage of LP-based algorithms you want to find the convex hull, $\text{conv}(\mathcal{F})$, of $\mathcal{F}$. If you can find $\text{conv}(\mathcal{F})$ and describe it compactly, then you can solve a mixed integer linear program with a linear programming solver. This is generally very difficult, so you must be satisfied with finding an approximation. Typically, the better the approximation, the more efficiently the LP-based branch-and-bound algorithm can perform.

As described in the section "Branch-and-Bound Algorithm" on page 336, the branch-and-bound algorithm begins by solving the linear programming relaxation over the polyhedron $\mathcal{Q}$. Clearly, $\mathcal{Q}$ contains the convex hull of the feasible region of the original integer program; that is, $\text{conv}(\mathcal{F}) \subseteq \mathcal{Q}$.

Cutting plane techniques are used to tighten the linear relaxation to better approximate $\text{conv}(\mathcal{F})$. Assume you are given a solution $\bar{x}$ to some intermediate linear relaxation during the branch-and-bound algorithm. A cut, or valid inequality ($\pi x \leq \pi^0$), is some half-space with the following characteristics:

- The half-space contains $\text{conv}(\mathcal{F})$; that is, every integer feasible solution is feasible for the cut ($\pi x \leq \pi^0, \forall x \in \mathcal{F}$).
- The half-space does not contain the current solution $\bar{x}$; that is, $\bar{x}$ is not feasible for the cut ($\pi \bar{x} > \pi^0$).

Cutting planes were first made popular by Dantzig, Fulkerson, and Johnson (1954) in their work on the traveling salesman problem. The two major classifications of cutting planes are *generic cuts* and *structured cuts*. Generic cuts are based solely on algebraic arguments and can be applied to any relaxation of any integer program. Structured cuts are specific to certain structures that can be found in some relaxations of the mixed integer linear program. These structures are automatically discovered during the cut initialization phase of the

MILP solver. Table 8.15 lists the various types of cutting planes that are built into the MILP solver. Included in each type are algorithms for numerous variations based on different relaxations and lifting techniques. For a survey of cutting plane techniques for mixed integer programming, see Marchand et al. (1999). For a survey of lifting techniques, see Atamturk (2004).

Table 8.15 Cutting Planes in the MILP Solver

Generic Cutting Planes	Structured Cutting Planes
Gomory mixed integer	Cliques
Lift-and-project	Flow cover
Mixed integer rounding	Flow path
Mixed lifted 0-1	Generalized upper bound cover
Zero-half	Implied bound
	Knapsack cover
	Multicommodity network flow

You can set levels for individual cuts by using the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIMPLIED=, CUTKNAPSACK=, CUTLAP=, CUTMILIFTED=, CUTMIR=, CUTMULTICOMMODITY=, and CUTZEROHALF= options. The valid levels for these options are listed in Table 8.14.

The cut level determines the internal strategy that is used by the MILP solver for generating the cutting planes. The strategy consists of several factors, including how frequently the cut search is called, the number of cuts allowed, and the aggressiveness of the search algorithms.

Sophisticated cutting planes, such as those included in the MILP solver, can take a great deal of CPU time. Usually, additional tightening of the relaxation helps speed up the overall process because it provides better bounds for the branch-and-bound tree and helps guide the LP solver toward integer solutions. In rare cases, shutting off cutting planes completely might lead to faster overall run times.

The default settings of the MILP solver have been tuned to work well for most instances. However, problem-specific expertise might suggest adjusting one or more of the strategies. These options give you that flexibility.

Primal Heuristics

Primal heuristics, an important component of the MILP solver in PROC OPTMODEL, are applied during the branch-and-bound algorithm. They are used to find integer feasible solutions early in the search tree, thereby improving the upper bound for a minimization problem. Primal heuristics play a role that is complementary to cutting planes in reducing the gap between the upper and lower bounds, thus reducing the size of the branch-and-bound tree.

Applying primal heuristics in the branch-and-bound algorithm assists in the following areas:

- finding a good upper bound early in the tree search (this can lead to earlier fathoming, resulting in fewer subproblems to be processed)
- locating a reasonably good feasible solution when that is sufficient (sometimes a reasonably good feasible solution is the best the solver can produce within certain time or resource limits)
- providing upper bounds for some bound-tightening techniques

The MILP solver implements several heuristic methodologies. Some algorithms, such as rounding and iterative rounding (diving) heuristics, attempt to construct an integer feasible solution by using fractional solutions to the continuous relaxation at each node of the branch-and-cut tree. Other algorithms start with an incumbent solution and attempt to find a better solution within a neighborhood of the current best solution.

The HEURISTICS= option enables you to control the level of primal heuristics that are applied by the MILP solver. This level determines how frequently primal heuristics are applied during the tree search. Some expensive heuristics might be disabled by the solver at less aggressive levels. Setting the HEURISTICS= option to a lower level also reduces the maximum number of iterations that are allowed in iterative heuristics. The valid values for this option are listed in Table 8.8.

Parallel Processing

You can run the branch-and-cut algorithm only in single-machine mode. In single-machine mode, the computation is executed by multiple threads on a single computer.

You can run the decomposition algorithm in either single-machine or distributed mode. In distributed mode, the computation is executed on multiple computing nodes in a distributed computing environment.

NOTE: Distributed mode requires SAS High-Performance Optimization.

You can specify options that control parallel processing in the PERFORMANCE statement, which is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Node Log

The following information about the status of the branch-and-bound algorithm is printed in the node log:

Node	indicates the sequence number of the current node in the search tree.
Active	indicates the current number of active nodes in the branch-and-bound tree.
Sols	indicates the number of feasible solutions found so far.
BestInteger	indicates the best upper bound (assuming minimization) found so far.
BestBound	indicates the best lower bound (assuming minimization) found so far.
Gap	indicates the relative gap between BestInteger and BestBound, displayed as a percentage. If the relative gap is larger than 1,000, then the absolute gap is displayed. If no active nodes remain, the value of Gap is 0.
Time	indicates the elapsed real time.

The LOGFREQ= option can be used to control the amount of information printed in the node log. By default a new entry is included in the log at the first node, at the last node, and at 100-node intervals. A new entry is also included each time a better integer solution is found. The LOGFREQ= option enables you to change the interval between entries in the node log. Figure 8.3 shows a sample node log.

Figure 8.3 Sample Node Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 10 variables (0 free, 0 fixed).
NOTE: The problem has 0 binary and 10 integer variables.
NOTE: The problem has 2 linear constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 20 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 2 variables and 0 constraints.
NOTE: The MILP presolver removed 4 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 8 variables, 2 constraints, and 16 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       3     85.0000000    158.0000000   46.20%       0
             0       1       3     85.0000000     88.0955497    3.51%       0
             0       1       3     85.0000000     88.0955497    3.51%       0
             0       1       3     85.0000000     88.0955497    3.51%       0
NOTE: The MILP presolver is applied again.
             0       1       3     85.0000000     88.0955497    3.51%       0
             0       1       3     85.0000000     88.0626822    3.48%       0
             0       1       3     85.0000000     87.8820655    3.28%       0
             0       1       3     85.0000000     87.8539763    3.25%       0
             0       1       3     85.0000000     87.7208690    3.10%       0
             0       1       3     85.0000000     87.7180302    3.10%       0
             0       1       3     85.0000000     87.7133502    3.09%       0
             0       1       3     85.0000000     87.7128245    3.09%       0
NOTE: The MILP solver added 2 cuts with 8 cut coefficients at the root.
             3       1       4     86.0000000     87.3854749    1.59%       0
             4       0       5     87.0000000     87.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 87.
```

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMODEL procedure causes the ODS table ProblemStatistics to be generated when the MILP solver is called. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 8.4 demonstrates the contents of the ODS table ProblemStatistics.

Figure 8.4 ODS Table ProblemStatistics

ProblemStatistics

Obs	Label1	cValue1	nValue1
1	Number of Constraint Matrix Nonzeros	8	8.000000
2	Maximum Constraint Matrix Coefficient	3	3.000000
3	Minimum Constraint Matrix Coefficient	1	1.000000
4	Average Constraint Matrix Coefficient	1.875	1.875000
5			.
6	Number of Objective Nonzeros	3	3.000000
7	Maximum Objective Coefficient	4	4.000000
8	Minimum Objective Coefficient	2	2.000000
9	Average Objective Coefficient	3	3.000000
10			.
11	Number of RHS Nonzeros	3	3.000000
12	Maximum RHS	7	7.000000
13	Minimum RHS	4	4.000000
14	Average RHS	5.3333333333	5.333333
15			.
16	Maximum Number of Nonzeros per Column	3	3.000000
17	Minimum Number of Nonzeros per Column	2	2.000000
18	Average Number of Nonzeros per Column	2.67	2.666667
19			.
20	Maximum Number of Nonzeros per Row	3	3.000000
21	Minimum Number of Nonzeros per Row	2	2.000000
22	Average Number of Nonzeros per Row	2.67	2.666667

The variable names in the ODS table ProblemStatistics are Label1, cValue1, and nValue1.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure defines a macro variable named _OROPTMODEL_. This variable contains a character string that indicates the status of the solver upon termination. The contents of the macro variable depend on which solver was invoked. For the MILP solver, the various terms of _OROPTMODEL_ are interpreted as follows.

STATUS
indicates the solver status at termination. It can take one of the following values:

OK	The solver terminated normally.
SYNTAX_ERROR	Syntax was used incorrectly.
DATA_ERROR	The input data was inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the solver.
IO_ERROR	A problem occurred in reading or writing data.

SEMANTIC_ERROR	An evaluation error, such as an invalid operand type, was found.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take one of the following values:

BAC	The branch-and-cut algorithm produced the solution data.
DECOMP	The decomposition algorithm produced the solution data.

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
OPTIMAL_AGAP	The solution is optimal within the absolute gap specified by the ABSOBJGAP= option.
OPTIMAL_RGAP	The solution is optimal within the relative gap specified by the RELOBJGAP= option.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances due to scaling or choice of small INTTOL= value.
TARGET	The solution is not worse than the target specified by the TARGET= option.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
BAD_PROBLEM_TYPE	The problem type is unsupported by solver.
SOLUTION_LIM	The solver reached the maximum number of solutions specified by the MAXSOLS= option.
NODE_LIM_SOL	The solver reached the maximum number of nodes specified by the MAXNODES= option and found a solution.
NODE_LIM_NOSOL	The solver reached the maximum number of nodes specified by the MAXNODES= option and did not find a solution.
TIME_LIM_SOL	The solver reached the execution time limit specified by the MAXTIME= option and found a solution.
TIME_LIM_NOSOL	The solver reached the execution time limit specified by the MAXTIME= option and did not find a solution.
ABORT_SOL	The solver was stopped by user but still found a solution.
ABORT_NOSOL	The solver was stopped by user and did not find a solution.
OUTMEM_SOL	The solver ran out of memory but still found a solution.

OUTMEM_NOSOL	The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
FAIL_SOL	The solver stopped due to errors but still found a solution.
FAIL_NOSOL	The solver stopped due to errors and did not find a solution.

OBJECTIVE
indicates the objective value obtained by the solver at termination.

RELATIVE_GAP
indicates the relative gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the MILP solver. The relative gap is equal to

$$| \text{BestInteger} - \text{BestBound} | / (1\text{E}{-}10 + | \text{BestBound} |)$$

ABSOLUTE_GAP
indicates the absolute gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the MILP solver. The absolute gap is equal to $| \text{BestInteger} - \text{BestBound} |$.

PRIMAL_INFEASIBILITY
indicates the maximum (absolute) violation of the primal constraints by the solution.

BOUND_INFEASIBILITY
indicates the maximum (absolute) violation by the solution of the lower or upper bounds (or both).

INTEGER_INFEASIBILITY
indicates the maximum (absolute) violation of the integrality of integer variables returned by the MILP solver.

BEST_BOUND
indicates the best bound on the objective function value at termination. A missing value indicates that the MILP solver was not able to obtain such a bound.

NODES
indicates the number of nodes enumerated by the MILP solver by using the branch-and-bound algorithm.

ITERATIONS
indicates the number of simplex iterations taken to solve the problem.

PRESOLVE_TIME
indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

Examples: MILP Solver

This section contains examples that illustrate the options and syntax of the MILP solver in PROC OPTMODEL. Example 8.1 illustrates the use of PROC OPTMODEL to solve an employee scheduling problem. Example 8.2 discusses a multicommodity transshipment problem with fixed charges. Example 8.3 demonstrates how to warm start the MILP solver. Example 8.4 shows the solution of an instance of the traveling salesman problem in PROC OPTMODEL. Other examples of mixed integer linear programs, along with example SAS code, are given in Chapter 13.

Example 8.1: Scheduling

The following example has been adapted from the example "A Scheduling Problem" in Chapter 4, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

Scheduling is a common application area in which mixed integer linear programming techniques are used. In this example, you have eight one-hour time slots in each of five days. You have to assign four employees to these time slots so that each slot is covered every day. You allow the employees to specify preference data for each slot on each day. In addition, the following constraints must be satisfied:

- Each employee has some time slots for which he or she is unavailable (OneEmpPerSlot).
- Each employee must have either time slot 4 or time slot 5 off for lunch (EmpMustHaveLunch).
- Each employee can work at most two time slots in a row (AtMost2ConSlots).
- Each employee can work only a specified number of hours in the week (WeeklyHoursLimit).

To formulate this problem, let i denote a person, j denote a time slot, and k denote a day. Then, let $x_{ijk} = 1$ if person i is assigned to time slot j on day k, and 0 otherwise. Let p_{ijk} denote the preference of person i for slot j on day k. Let h_i denote the number of hours in a week that person i will work. The formulation of this problem follows:

$$
\begin{array}{lllll}
\max & \displaystyle\sum_{ijk} p_{ijk} x_{ijk} & & & \\
\text{s.t.} & \displaystyle\sum_{i} x_{ijk} = 1 & \forall j,k & & \text{(OneEmpPerSlot)} \\
& x_{i4k} + x_{i5k} \le 1 & \forall i,k & & \text{(EmpMustHaveLunch)} \\
& x_{i,\ell,k} + x_{i,\ell+1,k} + x_{i,\ell+2,k} \le 2 & \forall i,k, \text{and } l \le 6 & & \text{(AtMost2ConSlots)} \\
& \displaystyle\sum_{jk} x_{ijk} \le h_i & \forall i & & \text{(WeeklyHoursLimit)} \\
& x_{ijk} = 0 & \forall i,j,k \text{ s.t. } p_{ijk} > 0 & & \\
& x_{ijk} \in \{0,1\} & \forall i,j,k & &
\end{array}
$$

The following data set preferences gives the preferences for each individual, time slot, and day. A 10 represents the most desirable time slot, and a 1 represents the least desirable time slot. In addition, a 0 indicates that the time slot is not available. The data set maxhours gives the maximum number of hours each employee can work per week.

```
data preferences;
   input name $ slot mon tue wed thu fri;
   datalines;
marc  1    10 10 10 10 10
marc  2     9  9  9  9  9
marc  3     8  8  8  8  8
marc  4     1  1  1  1  1
marc  5     1  1  1  1  1
marc  6     1  1  1  1  1
marc  7     1  1  1  1  1
marc  8     1  1  1  1  1
mike  1    10  9  8  7  6
mike  2    10  9  8  7  6
mike  3    10  9  8  7  6
mike  4    10  3  3  3  3
mike  5     1  1  1  1  1
mike  6     1  2  3  4  5
mike  7     1  2  3  4  5
mike  8     1  2  3  4  5
bill  1    10 10 10 10 10
bill  2     9  9  9  9  9
bill  3     8  8  8  8  8
bill  4     0  0  0  0  0
bill  5     1  1  1  1  1
bill  6     1  1  1  1  1
bill  7     1  1  1  1  1
bill  8     1  1  1  1  1
bob   1    10  9  8  7  6
bob   2    10  9  8  7  6
bob   3    10  9  8  7  6
bob   4    10  3  3  3  3
bob   5     1  1  1  1  1
bob   6     1  2  3  4  5
bob   7     1  2  3  4  5
bob   8     1  2  3  4  5
;

data maxhours;
   input name $ hour;
   datalines;
marc  20
mike  20
bill  20
bob   20
;
```

Using PROC OPTMODEL, you can model and solve the scheduling problem as follows:

```
proc optmodel;

   /* read in the preferences and max hours from the data sets */
   set <string,num> DailyEmployeeSlots;
   set <string>     Employees;

   set <num>    TimeSlots = (setof {<name,slot> in DailyEmployeeSlots} slot);
   set <string> WeekDays  = {"mon","tue","wed","thu","fri"};

   num WeeklyMaxHours{Employees};
   num PreferenceWeights{DailyEmployeeSlots,Weekdays};
   num NSlots = card(TimeSlots);

   read data preferences into DailyEmployeeSlots=[name slot]
        {day in Weekdays} <PreferenceWeights[name,slot,day] = col(day)>;
   read data maxhours into Employees=[name] WeeklyMaxHours=hour;

   /* declare the binary assignment variable x[i,j,k] */
   var Assign{<name,slot> in DailyEmployeeSlots, day in Weekdays} binary;

   /* for each p[i,j,k] = 0, fix x[i,j,k] = 0 */
   for {<name,slot> in DailyEmployeeSlots, day in Weekdays:
       PreferenceWeights[name,slot,day] = 0}
         fix Assign[name,slot,day] = 0;

   /* declare the objective function */
   max TotalPreferenceWeight =
      sum{<name,slot> in DailyEmployeeSlots, day in Weekdays}
         PreferenceWeights[name,slot,day] * Assign[name,slot,day];

   /* declare the constraints */
   con OneEmpPerSlot{slot in TimeSlots, day in Weekdays}:
      sum{name in Employees} Assign[name,slot,day] = 1;

   con EmpMustHaveLunch{name in Employees, day in Weekdays}:
      Assign[name,4,day] + Assign[name,5,day] <= 1;

   con AtMost2ConsSlots{name in Employees, start in 1..NSlots-2,
                        day in Weekdays}:
      Assign[name,start,day] + Assign[name,start+1,day]
            + Assign[name,start+2,day] <= 2 ;

   con WeeklyHoursLimit{name in Employees}:
      sum{slot in TimeSlots, day in Weekdays} Assign[name,slot,day]
           <= WeeklyMaxHours[name];

   /* solve the model */
   solve with milp;

   /* clean up the solution */
   for {<name,slot> in DailyEmployeeSlots, day in Weekdays}
      Assign[name,slot,day] = round(Assign[name,slot,day],1e-6);
```

```
   create data report from [name slot]={<name,slot> in DailyEmployeeSlots:
      max {day in Weekdays} Assign[name,slot,day] > 0}
         {day in Weekdays} <col(day)=(if Assign[name,slot,day] > 0
         then Assign[name,slot,day] else .)>;
quit;
```

The following statements demonstrate how to use the TABULATE procedure to display a schedule that shows how the eight time slots are covered for the week:

```
title 'Reported Solution';
proc format;
   value xfmt 1='  xxx   ';
run;
proc tabulate data=report;
   class name slot;
   var mon--fri;
   table (slot * name), (mon tue wed thu fri)*sum=' '*f=xfmt.
     /misstext=' ';
run;
```

The output from the preceding code is displayed in Output 8.1.1.

Output 8.1.1 Scheduling Reported Solution

Reported Solution

slot	name	mon	tue	wed	thu	fri
1	marc	xxx	xxx	xxx	xxx	xxx
2	bob	xxx				
	marc		xxx	xxx	xxx	xxx
3	bill				xxx	xxx
	mike	xxx	xxx	xxx		
4	bob		xxx	xxx	xxx	xxx
	mike	xxx				
5	bill		xxx	xxx	xxx	xxx
	bob	xxx				
6	bob				xxx	
	mike	xxx	xxx	xxx		xxx
7	bob	xxx	xxx	xxx		xxx
	mike				xxx	
8	bob	xxx	xxx	xxx		xxx
	mike				xxx	

Example 8.2: Multicommodity Transshipment Problem with Fixed Charges

The following example has been adapted from the example "A Multicommodity Transshipment Problem with Fixed Charges" in Chapter 4, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

This example illustrates the use of PROC OPTMODEL to generate a mixed integer linear program to solve a multicommodity network flow model with fixed charges. Consider a network with nodes N, arcs A, and a set C of commodities to be shipped between the nodes. The commodities are defined in the data set COMMODITY_DATA, as follows:

```
title 'Multicommodity Transshipment Problem with Fixed Charges';

data commodity_data;
   do c = 1 to 4;
      output;
   end;
run;
```

Shipping cost s_{ijc} is for each of the four commodities c across each of the arcs (i, j). In addition, there is a fixed charge f_{ij} for the use of each arc (i, j). The shipping costs and fixed charges are defined in the data set ARC_DATA, as follows:

```
data arc_data;
   input from $ to $ c1 c2 c3 c4 fx;
   datalines;
farm-a  Chicago 20 15 17 22 100
farm-b  Chicago 15 15 15 30  75
farm-c  Chicago 30 30 10 10 100
farm-a  StLouis 30 25 27 22 150
farm-c  StLouis 10 9 11 10   75
Chicago NY      75 75 75 75 200
StLouis NY      80 80 80 80 200
;
run;
```

The supply (positive numbers) or demand (negative numbers) d_{ic} at each of the nodes for each commodity c is shown in the data set SUPPLY_DATA, as follows:

```
data supply_data;
   input node $ sd1 sd2 sd3 sd4;
   datalines;
farm-a  100  100  40   .
farm-b  100  200  50  50
farm-c   40  100  75 100
NY     -150 -200 -50 -75
;
run;
```

Let x_{ijc} define the flow of commodity c across arc (i, j). Let $y_{ij} = 1$ if arc (i, j) is used, and 0 otherwise. Since the total flow on an arc (i, j) must be at most the total demand across all nodes $k \in N$, you can define the trivial upper bound u_{ijc} as

$$x_{ijc} \leq u_{ijc} = \sum_{k \in N | d_{kc} < 0} (-d_{kc})$$

This model can be represented using the following mixed integer linear program:

$$\begin{array}{lllll}
\min & \displaystyle\sum_{(i,j)\in A}\sum_{c\in C} s_{ijc}x_{ijc} + \sum_{(i,j)\in A} f_{ij}y_{ij} & & & \\
\text{s.t.} & \displaystyle\sum_{j\in N|(i,j)\in A} x_{ijc} - \sum_{j\in N|(j,i)\in A} x_{jic} & \leq & d_{ic} & \forall i \in N, c \in C \quad \text{(balance_con)} \\
& x_{ijc} & \leq & u_{ijc}y_{ij} & \forall (i,j) \in A, c \in C \quad \text{(fixed_charge_con)} \\
& x_{ijc} & \geq & 0 & \forall (i,j) \in A, c \in C \\
& y_{ij} \in \{0,1\} & & & \forall (i,j) \in A
\end{array}$$

Constraint (balance_con) ensures conservation of flow for both supply and demand. Constraint (fixed_charge_con) models the fixed charge cost by forcing $y_{ij} = 1$ if $x_{ijc} > 0$ for some commodity $c \in C$.

The PROC OPTMODEL statements follow:

```
proc optmodel;
   set COMMODITIES;
   read data commodity_data into COMMODITIES=[c];

   set <str,str> ARCS;
   num unit_cost {ARCS, COMMODITIES};
   num fixed_charge {ARCS};
   read data arc_data into ARCS=[from to] {c in COMMODITIES}
      <unit_cost[from,to,c]=col('c'||c)> fixed_charge=fx;
   print unit_cost fixed_charge;

   set <str> NODES = union {<i,j> in ARCS} {i,j};
   num supply {NODES, COMMODITIES} init 0;
   read data supply_data nomiss into [node] {c in COMMODITIES}
      <supply[node,c]=col('sd'||c)>;
   print supply;

   var AmountShipped {ARCS, c in COMMODITIES} >= 0 <= sum {i in NODES}
      max(supply[i,c],0);

   /* UseArc[i,j] = 1 if arc (i,j) is used, 0 otherwise */
   var UseArc {ARCS} binary;

   /* TotalCost = variable costs + fixed charges */
   min TotalCost = sum {<i,j> in ARCS, c in COMMODITIES}
      unit_cost[i,j,c] * AmountShipped[i,j,c]
```

```
      + sum {<i,j> in ARCS} fixed_charge[i,j] * UseArc[i,j];

   con flow_balance {i in NODES, c in COMMODITIES}:
      sum {<(i),j> in ARCS} AmountShipped[i,j,c] -
      sum {<j,(i)> in ARCS} AmountShipped[j,i,c] <= supply[i,c];

   /* if AmountShipped[i,j,c] > 0 then UseArc[i,j] = 1 */
   con fixed_charge_def {<i,j> in ARCS, c in COMMODITIES}:
      AmountShipped[i,j,c] <= AmountShipped[i,j,c].ub * UseArc[i,j];

   solve;

   print AmountShipped;

   create data solution from [from to commodity]={<i,j> in ARCS,
   c in COMMODITIES: AmountShipped[i,j,c].sol ne 0} amount=AmountShipped;
quit;
```

Although the PROC LP example used `M = 1.0e6` in the FIXED_CHARGE_DEF constraint that links the continuous variable to the binary variable, it is numerically preferable to use a smaller, data-dependent value. Here, the upper bound on `AmountShipped[i,j,c]` is used instead. This upper bound is calculated in the first VAR statement as the sum of all positive supplies for commodity c. The logical condition `AmountShipped[i,j,k].sol ne 0` in the CREATE DATA statement ensures that only the nonzero parts of the solution appear in the SOLUTION data set.

The problem summary, solution summary, and the output from the two PRINT statements are shown in Output 8.2.1.

Output 8.2.1 Multicommodity Transshipment Problem with Fixed Charges Solution Summary

Multicommodity Transshipment Problem with Fixed Charges

The OPTMODEL Procedure

[1]	[2]	[3]	unit_cost
Chicago	NY	1	75
Chicago	NY	2	75
Chicago	NY	3	75
Chicago	NY	4	75
StLouis	NY	1	80
StLouis	NY	2	80
StLouis	NY	3	80
StLouis	NY	4	80
farm-a	Chicago	1	20
farm-a	Chicago	2	15
farm-a	Chicago	3	17
farm-a	Chicago	4	22
farm-a	StLouis	1	30
farm-a	StLouis	2	25
farm-a	StLouis	3	27
farm-a	StLouis	4	22
farm-b	Chicago	1	15
farm-b	Chicago	2	15
farm-b	Chicago	3	15
farm-b	Chicago	4	30
farm-c	Chicago	1	30
farm-c	Chicago	2	30
farm-c	Chicago	3	10
farm-c	Chicago	4	10
farm-c	StLouis	1	10
farm-c	StLouis	2	9
farm-c	StLouis	3	11
farm-c	StLouis	4	10

[1]	[2]	fixed_charge
Chicago	NY	200
StLouis	NY	200
farm-a	Chicago	100
farm-a	StLouis	150
farm-b	Chicago	75
farm-c	Chicago	100
farm-c	StLouis	75

Output 8.2.1 *continued*

	supply			
	1	**2**	**3**	**4**
Chicago	0	0	0	0
NY	-150	-200	-50	-75
StLouis	0	0	0	0
farm-a	100	100	40	0
farm-b	100	200	50	50
farm-c	40	100	75	100

Problem Summary	
Objective Sense	Minimization
Objective Function	TotalCost
Objective Type	Linear
Number of Variables	35
Bounded Above	0
Bounded Below	0
Bounded Below and Above	35
Free	0
Fixed	0
Binary	7
Integer	0
Number of Constraints	52
Linear LE (<=)	52
Linear EQ (=)	0
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	112

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Output 8.2.1 *continued*

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	TotalCost
Solution Status	Optimal
Objective Value	42825
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	0
Best Bound	42825
Nodes	1
Iterations	45
Presolve Time	0.00
Solution Time	0.01

[1]	[2]	[3]	AmountShipped
Chicago	NY	1	110
Chicago	NY	2	100
Chicago	NY	3	50
Chicago	NY	4	75
StLouis	NY	1	40
StLouis	NY	2	100
StLouis	NY	3	0
StLouis	NY	4	0
farm-a	Chicago	1	10
farm-a	Chicago	2	0
farm-a	Chicago	3	0
farm-a	Chicago	4	0
farm-a	StLouis	1	0
farm-a	StLouis	2	0
farm-a	StLouis	3	0
farm-a	StLouis	4	0
farm-b	Chicago	1	100
farm-b	Chicago	2	100
farm-b	Chicago	3	0
farm-b	Chicago	4	0
farm-c	Chicago	1	0
farm-c	Chicago	2	0
farm-c	Chicago	3	50
farm-c	Chicago	4	75
farm-c	StLouis	1	40
farm-c	StLouis	2	100
farm-c	StLouis	3	0
farm-c	StLouis	4	0

Example 8.3: Facility Location

Consider the classic facility location problem. Given a set L of customer locations and a set F of candidate facility sites, you must decide on which sites to build facilities and assign coverage of customer demand to these sites so as to minimize cost. All customer demand d_i must be satisfied, and each facility has a demand capacity limit C. The total cost is the sum of the distances c_{ij} between facility j and its assigned customer i, plus a fixed charge f_j for building a facility at site j. Let $y_j = 1$ represent choosing site j to build a facility, and 0 otherwise. Also, let $x_{ij} = 1$ represent the assignment of customer i to facility j, and 0 otherwise. This model can be formulated as the following integer linear program:

$$
\begin{array}{lllll}
\min & \displaystyle\sum_{i\in L}\sum_{j\in F} c_{ij}x_{ij} + \sum_{j\in F} f_j y_j & & & \\
\text{s.t.} & \displaystyle\sum_{j\in F} x_{ij} & = 1 & \forall i \in L & \text{(assign\_def)} \\
& x_{ij} & \le y_j & \forall i \in L, j \in F & \text{(link)} \\
& \displaystyle\sum_{i\in L} d_i x_{ij} & \le C y_j & \forall j \in F & \text{(capacity)} \\
& x_{ij} \in \{0,1\} & & \forall i \in L, j \in F & \\
& y_j \in \{0,1\} & & \forall j \in F &
\end{array}
$$

Constraint (assign_def) ensures that each customer is assigned to exactly one site. Constraint (link) forces a facility to be built if any customer has been assigned to that facility. Finally, constraint (capacity) enforces the capacity limit at each site.

Consider also a variation of this same problem where there is no cost for building a facility. This problem is typically easier to solve than the original problem. For this variant, let the objective be

$$\min \sum_{i\in L}\sum_{j\in F} c_{ij}x_{ij}$$

First, construct a random instance of this problem by using the following DATA steps:

```
title 'Facility Location Problem';


%let NumCustomers  = 50;
%let NumSites      = 10;
%let SiteCapacity  = 35;
%let MaxDemand     = 10;
%let xmax          = 200;
%let ymax          = 100;
%let seed          = 938;

/* generate random customer locations */
data cdata(drop=i);
   length name $8;
   do i = 1 to &NumCustomers;
      name = compress('C'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
```

```
      demand = ranuni(&seed) * &MaxDemand;
      output;
   end;
run;

/* generate random site locations and fixed charge */
data sdata(drop=i);
   length name $8;
   do i = 1 to &NumSites;
      name = compress('SITE'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
      fixed_charge = 30 * (abs(&xmax/2-x) + abs(&ymax/2-y));
      output;
   end;
run;
```

The following PROC OPTMODEL statements first generate and solve the model with the no-fixed-charge variant of the cost function. Next, they solve the fixed-charge model. Note that the solution to the model with no fixed charge is feasible for the fixed-charge model and should provide a good starting point for the MILP solver. Use the PRIMALIN option to provide an incumbent solution (warm start).

```
proc optmodel;
   set <str> CUSTOMERS;
   set <str> SITES init {};
   /* x and y coordinates of CUSTOMERS and SITES */
   num x {CUSTOMERS union SITES};
   num y {CUSTOMERS union SITES};
   num demand {CUSTOMERS};
   num fixed_charge {SITES};
   /* distance from customer i to site j */
   num dist {i in CUSTOMERS, j in SITES}
       = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);
   read data cdata into CUSTOMERS=[name] x y demand;
   read data sdata into SITES=[name] x y fixed_charge;
   var Assign {CUSTOMERS, SITES} binary;
   var Build {SITES} binary;
   min CostNoFixedCharge
       = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j];
   min CostFixedCharge
       = CostNoFixedCharge + sum {j in SITES} fixed_charge[j] * Build[j];
   /* each customer assigned to exactly one site */
   con assign_def {i in CUSTOMERS}:
      sum {j in SITES} Assign[i,j] = 1;
   /* if customer i assigned to site j, then facility must be built at j */
   con link {i in CUSTOMERS, j in SITES}:
      Assign[i,j] <= Build[j];
   /* each site can handle at most &SiteCapacity demand */
   con capacity {j in SITES}:
      sum {i in CUSTOMERS} demand[i] * Assign[i,j] <=
         &SiteCapacity * Build[j];
   /* solve the MILP with no fixed charges */
   solve obj CostNoFixedCharge with milp / logfreq = 500;
```

```
   /* clean up the solution */
   for {i in CUSTOMERS, j in SITES} Assign[i,j] = round(Assign[i,j]);
   for {j in SITES} Build[j] = round(Build[j]);
   call symput('varcostNo',put(CostNoFixedCharge,6.1));
   /* create a data set for use by GPLOT */
   create data CostNoFixedCharge_Data from
      [customer site]={i in CUSTOMERS, j in SITES: Assign[i,j] = 1}
      xi=x[i] yi=y[i] xj=x[j] yj=y[j];
   /* solve the MILP, with fixed charges with warm start */
   solve obj CostFixedCharge with milp / primalin logfreq = 500;
   /* clean up the solution */
   for {i in CUSTOMERS, j in SITES} Assign[i,j] = round(Assign[i,j]);
   for {j in SITES} Build[j] = round(Build[j]);
   num varcost = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j].sol;
   num fixcost = sum {j in SITES} fixed_charge[j] * Build[j].sol;
   call symput('varcost', put(varcost,6.1));
   call symput('fixcost', put(fixcost,5.1));
   call symput('totalcost', put(CostFixedCharge,6.1));
   /* create a data set for use by GPLOT */
   create data CostFixedCharge_Data from
      [customer site]={i in CUSTOMERS, j in SITES: Assign[i,j] = 1}
      xi=x[i] yi=y[i] xj=x[j] yj=y[j];
quit;
```

The information printed in the log for the no-fixed-charge model is displayed in Output 8.3.1.

Output 8.3.1 OPTMODEL Log for Facility Location with No Fixed Charges

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 510 variables (0 free, 0 fixed).
NOTE: The problem has 510 binary and 0 integer variables.
NOTE: The problem has 560 linear constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 10 variables and 500 constraints.
NOTE: The MILP presolver removed 1010 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 500 variables, 60 constraints, and 1000
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound       Gap    Time
             0       1       2    972.1737321              0    972.2       0
             0       1       2    972.1737321    961.2403449    1.14%       0
             0       1       2    972.1737321    961.2403449    1.14%       0
             0       1       2    972.1737321    961.2403449    1.14%       0
NOTE: The MILP presolver is applied again.
             0       1       4    966.4832160    962.9120804    0.37%       0
             0       1       5    966.4832160    966.4832160    0.00%       0
             0       0       5    966.4832160    966.4832160    0.00%       0
NOTE: The MILP solver added 0 cuts with 0 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 966.48321599.
```

The results from the warm start approach are shown in Output 8.3.2.

Output 8.3.2 OPTMODEL Log for Facility Location with Fixed Charges, Using Warm Start

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 510 variables (0 free, 0 fixed).
NOTE: The problem uses 1 implicit variables.
NOTE: The problem has 510 binary and 0 integer variables.
NOTE: The problem has 560 linear constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 510 variables, 560 constraints, and 2010
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       3  16070.0150023              0    16070       0
             0       1       3  16070.0150023   9946.2514269   61.57%       0
             0       1       3  16070.0150023   9962.4849932   61.31%       0
             0       1       3  16070.0150023   9971.5893075   61.16%       0
             0       1       3  16070.0150023   9974.5588580   61.11%       0
             0       1       3  16070.0150023   9978.1322942   61.05%       0
             0       1       3  16070.0150023   9978.3312183   61.05%       0
             0       1       3  16070.0150023   9980.4930282   61.01%       0
             0       1       3  16070.0150023   9981.2701907   61.00%       0
             0       1       4  16009.9864088   9981.2701907   60.40%       0
NOTE: The MILP solver added 20 cuts with 631 cut coefficients at the root.
           207     181       5  14940.8468978  10254.0277196   45.71%       0
           244     203       6  11574.7346191  10366.4971368   11.66%       0
           304     250       7  11200.4907322  10384.6282929    7.86%       0
           464     318       8  10952.9510709  10474.3603655    4.57%       0
           500     339       8  10952.9510709  10548.0238535    3.84%       0
           548     368       9  10951.0805543  10548.0238535    3.82%       0
           560     374      10  10951.0805542  10548.0238535    3.82%       0
           642       6      11  10950.0345546  10946.4429359    0.03%       0
           660       0      12  10948.4603464  10948.4603464    0.00%       0
NOTE: Optimal.
NOTE: Objective = 10948.460346.
```

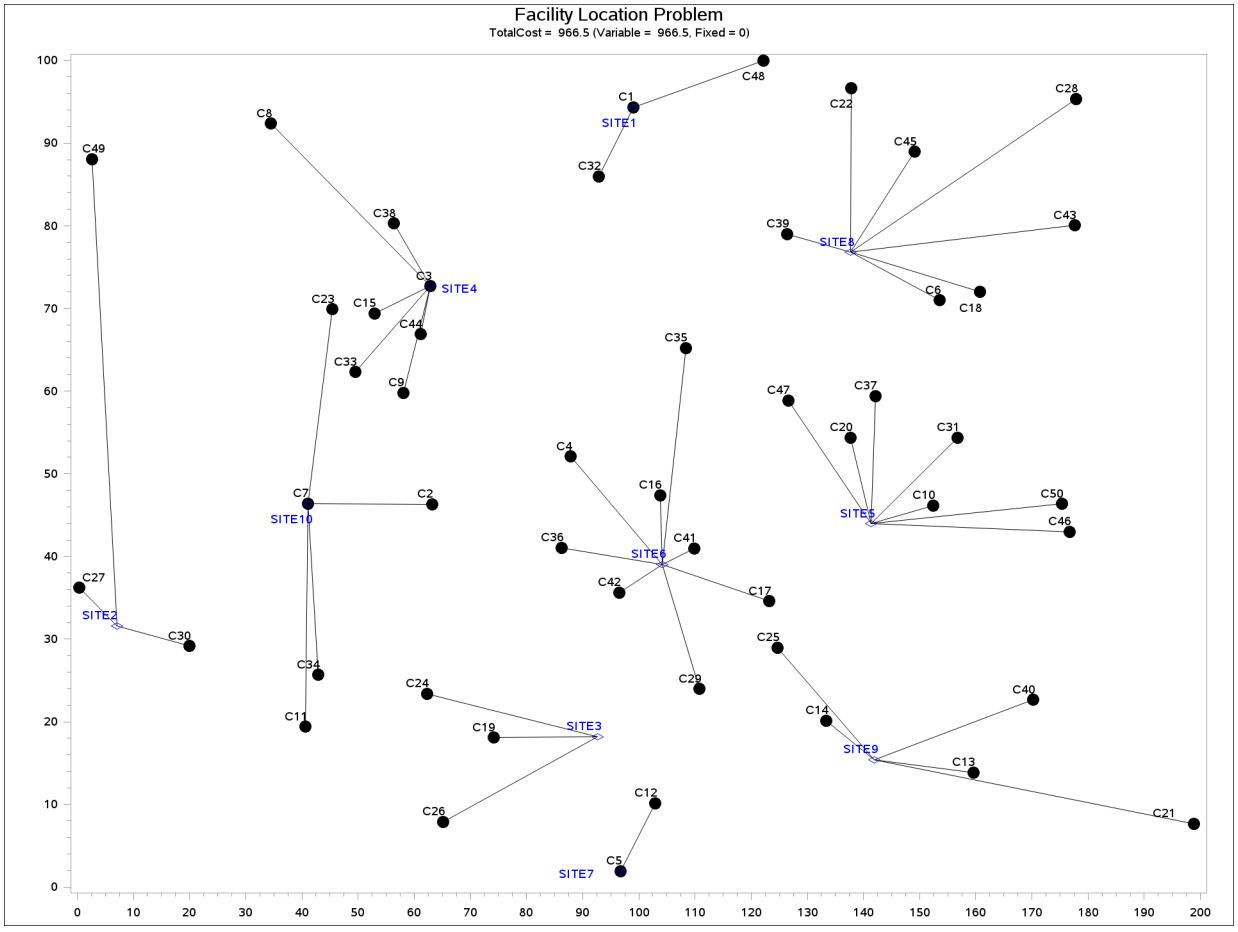
The following two SAS programs produce a plot of the solutions for both variants of the model, using data sets produced by PROC OPTMODEL:

```
title1 h=1.5 "Facility Location Problem";
title2 "TotalCost = &varcostNo (Variable = &varcostNo, Fixed = 0)";

data csdata;
   set cdata(rename=(y=cy)) sdata(rename=(y=sy));
run;
```

```
/* create Annotate data set to draw line between customer and assigned site */
%annomac;
data anno(drop=xi yi xj yj);
   %SYSTEM(2, 2, 2);
   set CostNoFixedCharge_Data(keep=xi yi xj yj);
   %LINE(xi, yi, xj, yj, *, 1, 1);
run;

proc gplot data=csdata anno=anno;
   axis1 label=none order=(0 to &xmax by 10);
   axis2 label=none order=(0 to &ymax by 10);
   symbol1 value=dot interpol=none
      pointlabel=("#name" nodropcollisions height=1) cv=black;
   symbol2 value=diamond interpol=none
      pointlabel=("#name" nodropcollisions color=blue height=1) cv=blue;
   plot cy*x sy*x / overlay haxis=axis1 vaxis=axis2;
run;
quit;
```

The output of the first program is shown in Output 8.3.3.

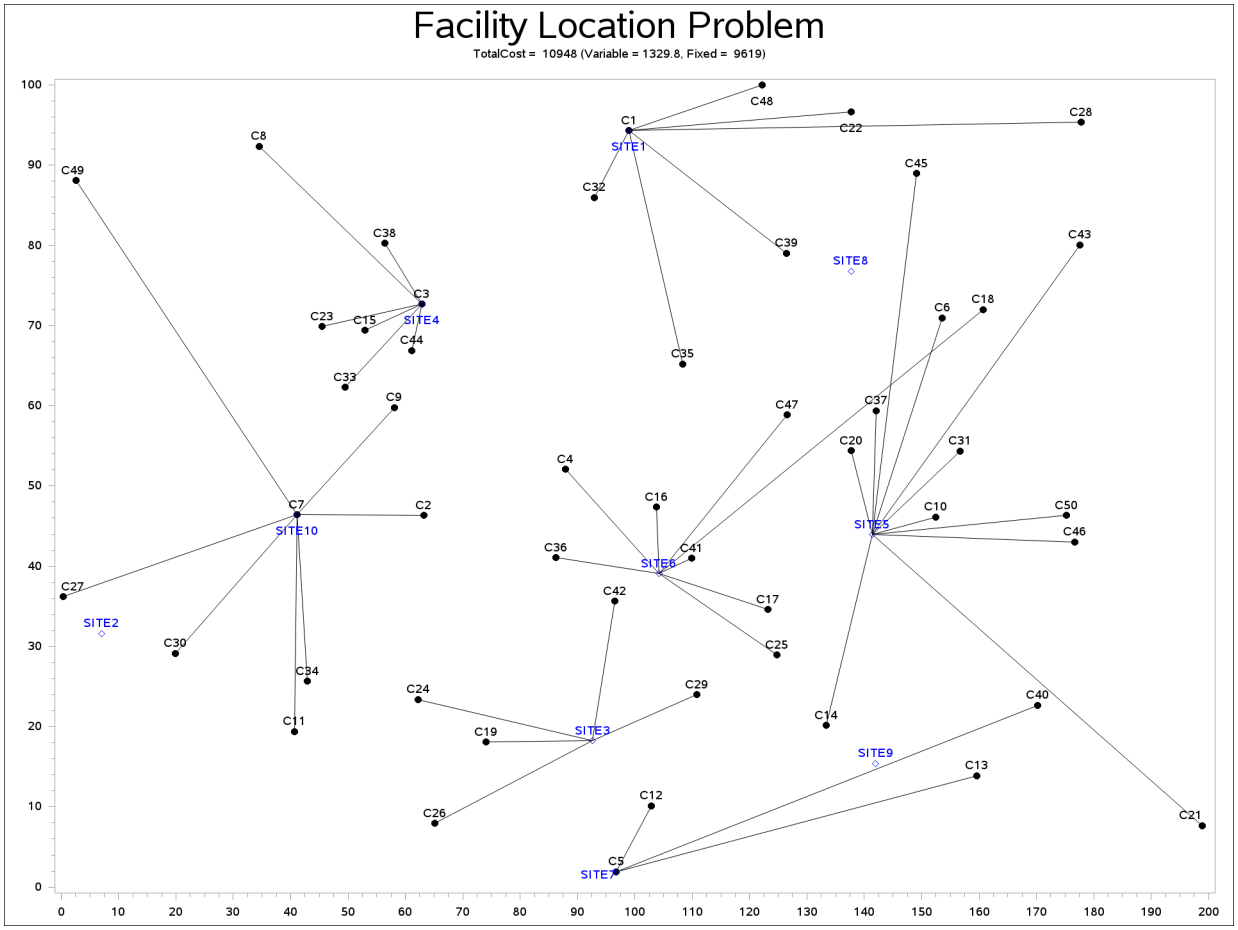
Output 8.3.3 Solution Plot for Facility Location with No Fixed Charges



The output of the second program is shown in Output 8.3.4.

```
title1 "Facility Location Problem";
title2 "TotalCost = &totalcost (Variable = &varcost, Fixed = &fixcost)";

/* create Annotate data set to draw line between customer and assigned site */
data anno(drop=xi yi xj yj);
   %SYSTEM(2, 2, 2);
   set CostFixedCharge_Data(keep=xi yi xj yj);
   %LINE(xi, yi, xj, yj, *, 1, 1);
run;

proc gplot data=csdata anno=anno;
   axis1 label=none order=(0 to &xmax by 10);
   axis2 label=none order=(0 to &ymax by 10);
   symbol1 value=dot interpol=none
      pointlabel=("#name" nodropcollisions height=1) cv=black;
   symbol2 value=diamond interpol=none
      pointlabel=("#name" nodropcollisions color=blue height=1) cv=blue;
   plot cy*x sy*x / overlay haxis=axis1 vaxis=axis2;
run;
quit;
```

Output 8.3.4 Solution Plot for Facility Location with Fixed Charges



The economic trade-off for the fixed-charge model forces you to build fewer sites and push more demand to each site.

It is possible to expedite the solution of the fixed-charge facility location problem by choosing appropriate branching priorities for the decision variables. Recall that for each site j, the value of the variable y_j determines whether or not a facility is built on that site. Suppose you decide to branch on the variables y_j before the variables x_{ij}. You can set a higher branching priority for y_j by using the .priority suffix for the Build variables in PROC OPTMODEL, as follows:

```
for{j in SITES} Build[j].priority=10;
```

Setting higher branching priorities for certain variables is not guaranteed to speed up the MILP solver, but it can be helpful in some instances. The following program creates and solves an instance of the facility location problem, giving higher priority to the variables y_j. The LOGFREQ= option is used to abbreviate the node log.

```
%let NumCustomers  = 45;
%let NumSites      = 8;
%let SiteCapacity  = 35;
%let MaxDemand     = 10;
%let xmax          = 200;
%let ymax          = 100;
%let seed          = 2345;

/* generate random customer locations */
data cdata(drop=i);
   length name $8;
   do i = 1 to &NumCustomers;
      name = compress('C'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
      demand = ranuni(&seed) * &MaxDemand;
      output;
   end;
run;

/* generate random site locations and fixed charge */
data sdata(drop=i);
length name $8;
   do i = 1 to &NumSites;
      name = compress('SITE'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
      fixed_charge = (abs(&xmax/2-x) + abs(&ymax/2-y)) / 2;
      output;
   end;
run;
```

```
proc optmodel;
   set <str> CUSTOMERS;
   set <str> SITES init {};

   /* x and y coordinates of CUSTOMERS and SITES */
   num x {CUSTOMERS union SITES};
   num y {CUSTOMERS union SITES};
   num demand {CUSTOMERS};
   num fixed_charge {SITES};

   /* distance from customer i to site j */
   num dist {i in CUSTOMERS, j in SITES}
       = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);

   read data cdata into CUSTOMERS=[name] x y demand;
   read data sdata into SITES=[name] x y fixed_charge;

   var Assign {CUSTOMERS, SITES} binary;
   var Build {SITES} binary;

   min CostFixedCharge
       = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j]
         + sum {j in SITES} fixed_charge[j] * Build[j];

   /* each customer assigned to exactly one site */
   con assign_def {i in CUSTOMERS}:
      sum {j in SITES} Assign[i,j] = 1;

   /* if customer i assigned to site j, then facility must be built at j */
   con link {i in CUSTOMERS, j in SITES}:
      Assign[i,j] <= Build[j];

   /* each site can handle at most &SiteCapacity demand */
   con capacity {j in SITES}:
      sum {i in CUSTOMERS} demand[i] * Assign[i,j] <= &SiteCapacity * Build[j];

   /* assign priority to Build variables (y) */
   for{j in SITES} Build[j].priority=10;

   /* solve the MILP with fixed charges, using branching priorities */
   solve obj CostFixedCharge with milp / logfreq=1000;
quit;
```

The resulting output is shown in Output 8.3.5.

Output 8.3.5 PROC OPTMODEL Log for Facility Location with Branching Priorities

```
NOTE: There were 45 observations read from the data set WORK.CDATA.
NOTE: There were 8 observations read from the data set WORK.SDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 368 variables (0 free, 0 fixed).
NOTE: The problem has 368 binary and 0 integer variables.
NOTE: The problem has 413 linear constraints (368 LE, 45 EQ, 0 GE, 0 range).
NOTE: The problem has 1448 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 368 variables, 413 constraints, and 1448
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound       Gap    Time
             0       1       3   2823.1827978              0   2823.2       0
             0       1       3   2823.1827978   1727.0208789   63.47%       0
             0       1       3   2823.1827978   1756.0637224   60.77%       0
             0       1       5   1906.4633474   1764.3961986    8.05%       0
             0       1       5   1906.4633474   1772.0991932    7.58%       0
             0       1       5   1906.4633474   1784.2955379    6.85%       0
             0       1       5   1906.4633474   1788.0570704    6.62%       0
             0       1       5   1906.4633474   1788.0570704    6.62%       0
NOTE: The MILP presolver is applied again.
             0       1       5   1906.4633474   1788.0570704    6.62%       0
             0       1       5   1906.4633474   1788.0570704    6.62%       0
             0       1       5   1906.4633474   1788.0570704    6.62%       0
             0       1       5   1906.4633474   1788.0570704    6.62%       0
             0       1       5   1906.4633474   1788.0570704    6.62%       0
             0       1       5   1906.4633474   1788.1857838    6.61%       0
             0       1       5   1906.4633474   1793.0963734    6.32%       0
             0       1       5   1906.4633474   1793.5540695    6.30%       0
             0       1       5   1906.4633474   1793.6034154    6.29%       0
             0       1       5   1906.4633474   1793.7200171    6.29%       0
             0       1       5   1906.4633474   1793.7651419    6.28%       0
             0       1       6   1903.8518326   1793.7651419    6.14%       0
NOTE: The MILP solver added 28 cuts with 666 cut coefficients at the root.
           209     182       7   1826.3615927   1800.9334244    1.41%       0
           816      46       8   1823.4483963   1805.5532343    0.99%       0
          1000     195       8   1823.4483963   1808.1519690    0.85%       0
          1390     397       9   1823.4483963   1811.0458067    0.68%       0
          1845     352      10   1819.9124343   1814.9424888    0.27%       1
          2000     153      10   1819.9124343   1817.0347069    0.16%       1
          2167       7      10   1819.9124343   1819.7338302    0.01%       1
NOTE: Optimal within relative gap.
NOTE: Objective = 1819.9124343.
```

Example 8.4: Traveling Salesman Problem

The traveling salesman problem (TSP) is that of finding a minimum cost *tour* in an undirected graph G with vertex set $V = \{1, \ldots, |V|\}$ and edge set E. A tour is a connected subgraph for which each vertex has degree two. The goal is then to find a tour of minimum total cost, where the total cost is the sum of the costs of the edges in the tour. With each edge $e \in E$ we associate a binary variable x_e, which indicates whether edge e is part of the tour, and a cost $c_e \in \mathbb{R}$. Let $\delta(S) = \{\{i, j\} \in E \mid i \in S,\ j \notin S\}$. Then an integer linear programming (ILP) formulation of the TSP is as follows:

$$
\begin{array}{lllll}
\min & \displaystyle\sum_{e \in E} c_e x_e & & & \\
\text{s.t.} & \displaystyle\sum_{e \in \delta(i)} x_e = 2 & \forall i \in V & & \text{(two\_match)} \\
& \displaystyle\sum_{e \in \delta(S)} x_e \geq 2 & \forall S \subset V,\ 2 \leq |S| \leq |V| - 1 & & \text{(subtour\_elim)} \\
& x_e \in \{0, 1\} & \forall e \in E & &
\end{array}
$$

The equations (two_match) are the *matching constraints*, which ensure that each vertex has degree two in the subgraph, while the inequalities (subtour_elim) are known as the *subtour elimination constraints* (SECs) and enforce connectivity.

Since there is an exponential number $O(2^{|V|})$ of SECs, it is impossible to explicitly construct the full TSP formulation for large graphs. An alternative formulation of polynomial size was introduced by Miller, Tucker, and Zemlin (1960) (MTZ):

$$
\begin{array}{llllll}
\min & \displaystyle\sum_{(i,j) \in E} c_{ij} x_{ij} & & & & \\
\text{s.t.} & \displaystyle\sum_{j \in V} x_{ij} = 1 & & \forall i \in V & & \text{(assign\_i)} \\
& \displaystyle\sum_{i \in V} x_{ij} = 1 & & \forall j \in V & & \text{(assign\_j)} \\
& u_i - u_j + 1 \leq (|V| - 1)(1 - x_{ij}) & & \forall (i, j) \in V, i \neq 1, j \neq 1 & & \text{(mtz)} \\
& 2 \leq u_i \leq |V| & & \forall i \in \{2, .., |V|\}, & & \\
& x_{ij} \in \{0, 1\} & & \forall (i, j) \in E & &
\end{array}
$$

This formulation uses a directed graph. Constraints (assign_i) and (assign_j) now enforce that each vertex has degree two (one edge in, one edge out). The MTZ constraints (mtz) enforce that no subtours exist.

TSPLIB, located at http://elib.zib.de/pub/Packages/mp-testdata/tsp/tsplib/tsplib.html, is a set of benchmark instances for the TSP. The following DATA step converts a TSPLIB instance of type EUC_2D into a SAS data set that contains the coordinates of the vertices:

```
/* convert the TSPLIB instance into a data set */
data tspData(drop=H);
   infile "st70.tsp";
   input H $1. @;
   if H not in ('N','T','C','D','E');
   input @1 var1-var3;
run;
```

The following PROC OPTMODEL statements attempt to solve the TSPLIB instance st70.tsp by using the MTZ formulation:

```
/* direct solution using the MTZ formulation */
proc optmodel;
   set VERTICES;
   set EDGES = {i in VERTICES, j in VERTICES: i ne j};
   num xc {VERTICES};
   num yc {VERTICES};
   /* read in the instance and customer coordinates (xc, yc) */
   read data tspData into VERTICES=[_n_] xc=var2 yc=var3;
   /* the cost is the euclidean distance rounded to the nearest integer */
   num c {<i,j> in EDGES}
       init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);
   var x {EDGES} binary;
   var u {i in 2..card(VERTICES)} >= 2 <= card(VERTICES);
   /* each vertex has exactly one in-edge and one out-edge */
   con assign_i {i in VERTICES}:
       sum {j in VERTICES: i ne j} x[i,j] = 1;
   con assign_j {j in VERTICES}:
       sum {i in VERTICES: i ne j} x[i,j] = 1;
   /* minimize the total cost */
   min obj
       = sum {<i,j> in EDGES} (if i > j then c[i,j] else c[j,i]) * x[i,j];
   /* no subtours */
   con mtz {<i,j> in EDGES : (i ne 1) and (j ne 1)}:
       u[i] - u[j] + 1 <= (card(VERTICES) - 1) * (1 - x[i,j]);
   solve with milp / maxtime = 600;
quit;
```

It is well known that the MTZ formulation is much weaker than the subtour formulation. The exponential number of SECs makes it impossible, at least in large instances, to use a direct call to the MILP solver with the subtour formulation. For this reason, if you want to solve the TSP with one SOLVE statement, you must use the MTZ formulation and rely strictly on generic cuts and heuristics. Except for very small instances, this is unlikely to be a good approach.

A much more efficient way to tackle the TSP is to dynamically generate the subtour inequalities as *cuts*. Typically this is done by solving the LP relaxation of the two-matching problem, finding violated subtour cuts, and adding them iteratively. The problem of finding violated cuts is known as the *separation problem*. In this case, the separation problem takes the form of a minimum cut problem, which is nontrivial to implement efficiently. Therefore, for the sake of illustration, an integer program is solved at each step of the process.

The initial formulation of the TSP is the integral two-matching problem. You solve this by using PROC OPTMODEL to obtain an integral matching, which is not necessarily a tour. In this case, the separation problem is trivial. If the solution is a connected graph, then it is a tour, so the problem is solved. If the solution is a disconnected graph, then each component forms a violated subtour constraint. These constraints are added to the formulation, and the integer program is solved again. This process is repeated until the solution defines a tour.

The following PROC OPTMODEL statements solve the TSP by using the subtour formulation and iteratively adding subtour constraints:

```
/* iterative solution using the subtour formulation */
proc optmodel;
   set VERTICES;
   set EDGES = {i in VERTICES, j in VERTICES: i > j};
   num xc {VERTICES};
   num yc {VERTICES};

   num numsubtour init 0;
   set SUBTOUR {1..numsubtour};

   /* read in the instance and customer coordinates (xc, yc) */
   read data tspData into VERTICES=[var1] xc=var2 yc=var3;

   /* the cost is the euclidean distance rounded to the nearest integer */
   num c {<i,j> in EDGES}
       init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);

   var x {EDGES} binary;

   /* minimize the total cost */
   min obj =
       sum {<i,j> in EDGES} c[i,j] * x[i,j];

   /* each vertex has exactly one in-edge and one out-edge */
   con two_match {i in VERTICES}:
       sum {j in VERTICES: i > j} x[i,j]
     + sum {j in VERTICES: i < j} x[j,i] = 2;

   /* no subtours (these constraints are generated dynamically) */
   con subtour_elim {s in 1..numsubtour}:
       sum {<i,j> in EDGES: (i in SUBTOUR[s] and j not in SUBTOUR[s])
          or (i not in SUBTOUR[s] and j in SUBTOUR[s])} x[i,j] >= 2;

   /* this starts the algorithm to find violated subtours */
   set <num,num> EDGES1;
   set INITVERTICES = setof{<i,j> in EDGES1} i;
   set VERTICES1;
   set NEIGHBORS;
   set <num,num> CLOSURE;
   num component {INITVERTICES};
   num numcomp  init 2;
   num iter     init 1;
   num numiters init 1;
   set ITERS = 1..numiters;
   num sol {ITERS, EDGES};

   /* initial solve with just matching constraints */
   solve;
   call symput(compress('obj'||put(iter,best.)),
              trim(left(put(round(obj),best.))));
   for {<i,j> in EDGES} sol[iter,i,j] = round(x[i,j]);

   /* while the solution is disconnected, continue */
```

```
   do while (numcomp > 1);
      iter = iter + 1;

      /* find connected components of support graph   */
      EDGES1 = {<i,j> in EDGES: round(x[i,j].sol) = 1};
      EDGES1 = EDGES1 union {setof {<i,j> in EDGES1} <j,i>};
      VERTICES1 = INITVERTICES;
      CLOSURE = EDGES1;
      for {i in INITVERTICES} component[i] = 0;
      for {i in VERTICES1} do;
         NEIGHBORS = slice(<i,*>,CLOSURE);
         CLOSURE = CLOSURE union (NEIGHBORS cross NEIGHBORS);
      end;

      numcomp = 0;
      do while (card(VERTICES1) > 0);
         numcomp = numcomp + 1;
         for {i in VERTICES1} do;
            NEIGHBORS = slice(<i,*>,CLOSURE);
            for {j in NEIGHBORS} component[j] = numcomp;
            VERTICES1 = VERTICES1 diff NEIGHBORS;
            leave;
         end;
      end;

      if numcomp = 1 then leave;
      numiters = iter;
      numsubtour = numsubtour + numcomp;
      for {comp in 1..numcomp} do;
         SUBTOUR[numsubtour-numcomp+comp]
           = {i in VERTICES: component[i] = comp};
      end;

      solve;
      call symput(compress('obj'||put(iter,best.)),
                  trim(left(put(round(obj),best.))));
      for {<i,j> in EDGES} sol[iter,i,j] = round(x[i,j]);
   end;

   /* create a data set for use by gplot */
   create data solData from
      [iter i j]={it in ITERS, <i,j> in EDGES: sol[it,i,j] = 1}
      xi=xc[i] yi=yc[i] xj=xc[j] yj=yc[j];
   call symput('numiters',put(numiters,best.));
quit;
```

You can generate plots of the solution and objective value at each stage by using the following statements:

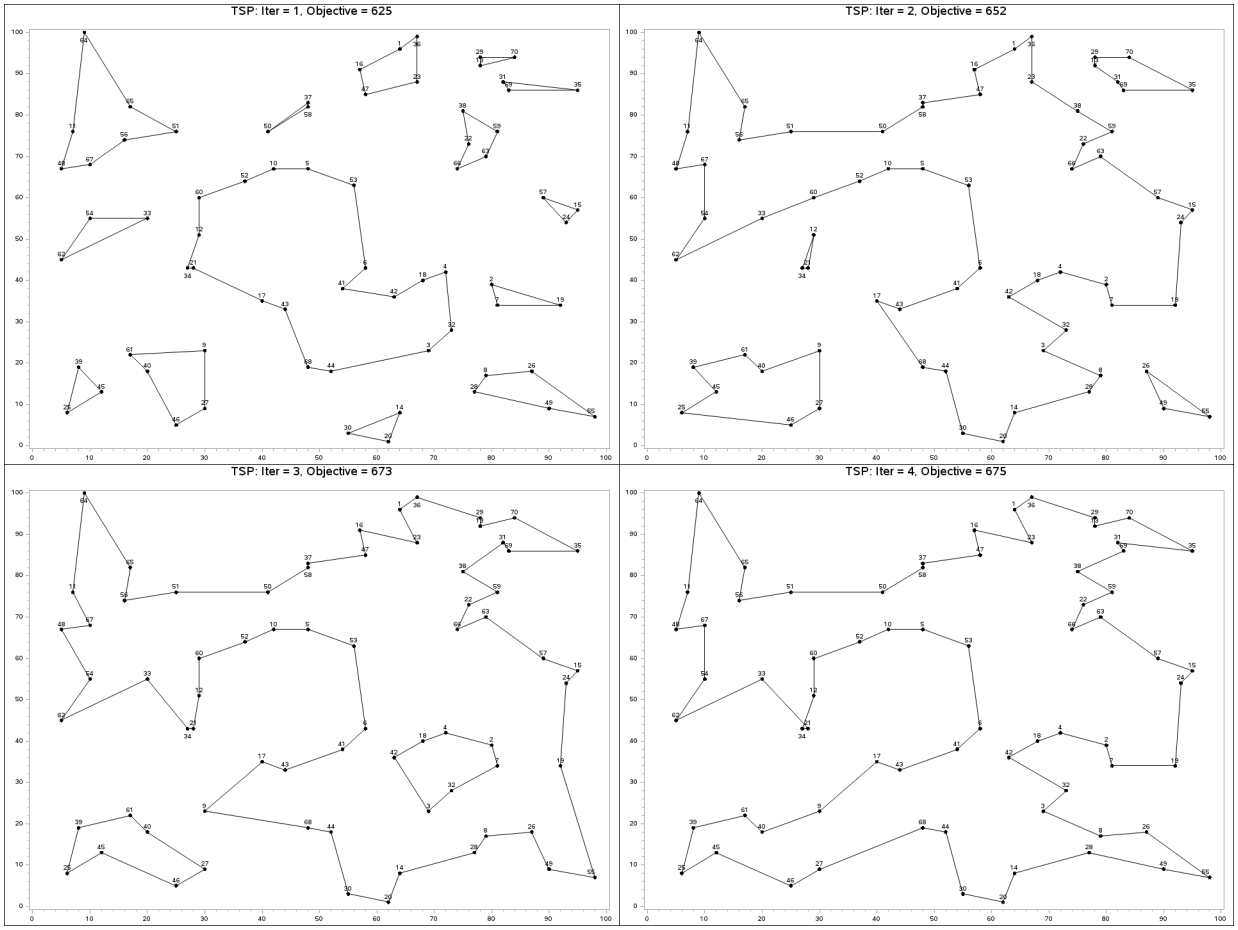
```
%macro plotTSP;
   %annomac;
   %do i = 1 %to &numiters;
      /* create annotate data set to draw subtours */
      data anno(drop=iter xi yi xj yj);
         %SYSTEM(2, 2, 2);
         set solData(keep=iter xi yi xj yj);
         where iter = &i;
         %LINE(xi, yi, xj, yj, *, 1, 1);
      run;

      title1 h=2 "TSP: Iter = &i, Objective = &&obj&i";
      title2;

      proc gplot data=tspData anno=anno;
         axis1 label=none;
         symbol1 value=dot interpol=none
         pointlabel=("#var1" nodropcollisions height=1) cv=black;
         plot var3*var2 / haxis=axis1 vaxis=axis1;
      run;
      quit;
   %end;
%mend plotTSP;

%plotTSP;
```

The plot in Output 8.4.1 shows the solution and objective value at each stage. Notice that each stage restricts some subset of subtours. When you reach the final stage, you have a valid tour.

NOTE: An alternative way of approaching the TSP is to use a genetic algorithm. See the "Examples" section in Chapter 4, "The GA Procedure" (*SAS/OR User's Guide: Local Search Optimization*), for an example of how to use PROC GA to solve the TSP.

NOTE: See the "Examples" section in Chapter 2, "The OPTNET Procedure" (*SAS/OR User's Guide: Network Optimization Algorithms*), for an example of how to use PROC OPTNET to solve the TSP.

Output 8.4.1 Traveling Salesman Problem Iterative Solution



References

Achterberg, T., Koch, T., and Martin, A. (2005). "Branching Rules Revisited." *Operations Research Letters* 33:42–54.

Andersen, E. D., and Andersen, K. D. (1995). "Presolving in Linear Programming." *Mathematical Programming* 71:221–245.

Atamturk, A. (2004). "Sequence Independent Lifting for Mixed-Integer Programming." *Operations Research* 52:487–490.

Dantzig, G. B., Fulkerson, R., and Johnson, S. M. (1954). "Solution of a Large-Scale Traveling Salesman Problem." *Operations Research* 2:393–410.

Gondzio, J. (1997). "Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method." *INFORMS Journal on Computing* 9:73–91.

Land, A. H., and Doig, A. G. (1960). "An Automatic Method for Solving Discrete Programming Problems." *Econometrica* 28:497–520.

Linderoth, J. T., and Savelsbergh, M. W. P. (1998). "A Computational Study of Search Strategies for Mixed Integer Programming." *INFORMS Journal on Computing* 11:173–187.

Marchand, H., Martin, A., Weismantel, R., and Wolsey, L. (1999). "Cutting Planes in Integer and Mixed Integer Programming." DP 9953, CORE, Université Catholique de Louvain.

Miller, C. E., Tucker, A. W., and Zemlin, R. A. (1960). "Integer Programming Formulations of Traveling Salesman Problems." *Journal of the Association for Computing Machinery* 7:326–329.

Ostrowski, J. (2008). *Symmetry in Integer Programming*. Ph.D. diss., Lehigh University.

Savelsbergh, M. W. P. (1994). "Preprocessing and Probing Techniques for Mixed Integer Programming Problems." *ORSA Journal on Computing* 6:445–454.

Chapter 9
The Network Solver

Contents

Overview: Network Solver **376**
Getting Started: Network Solver **376**
Syntax: Network Solver **381**
Functional Summary 382
SOLVE WITH NETWORK Statement 386
General Options 386
Input and Output Options 387
Algorithm Options 390
Details: Network Solver **398**
Input Data for the Network Solver 398
Solving over Subsets of Nodes and Links (Filters) 402
Numeric Limitations 406
Biconnected Components and Articulation Points 407
Clique 410
Connected Components 413
Cycle 418
Linear Assignment (Matching) 423
Minimum-Cost Network Flow 424
Minimum Cut 430
Minimum Spanning Tree 434
Shortest Path 436
Transitive Closure 449
Traveling Salesman Problem 452
Macro Variable _OROPTMODEL_ 458
Examples: Network Solver **462**
Example 9.1: Articulation Points in a Terrorist Network 462
Example 9.2: Cycle Detection for Kidney Donor Exchange 464
Example 9.3: Linear Assignment Problem for Minimizing Swim Times 468
Example 9.4: Linear Assignment Problem, Sparse Format versus Dense Format 471
Example 9.5: Minimum Spanning Tree for Computer Network Topology 474
Example 9.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System 476
Example 9.7: Traveling Salesman Tour through US Capital Cities 479
References **485**

Overview: Network Solver

The network solver includes a number of graph theory, combinatorial optimization, and network analysis algorithms. The algorithm classes are listed in Table 9.1.

Table 9.1 Algorithm Classes in the Network solver

Algorithm Class	SOLVE WITH NETWORK Option
Biconnected components	BICONCOMP
Maximal cliques	CLIQUE=
Connected components	CONCOMP
Cycle detection	CYCLE=
Linear assignment (matching)	LINEAR_ASSIGNMENT
Minimum-cost network flow	MINCOSTFLOW
Minimum cut	MINCUT=
Minimum spanning tree	MINSPANTREE
Shortest path	SHORTPATH=
Transitive closure	TRANSITIVE_CLOSURE
Traveling salesman	TSP=

You can use the network solver to analyze relationships between entities. These relationships are typically defined by using a *graph*. A graph, $G = (N, A)$, is defined over a set N of nodes, and a set A of links. A *node* is an abstract representation of some entity (or object), and an *arc* defines some relationship (or connection) between two nodes. The terms *node* and *vertex* are often interchanged in describing an entity. The term *arc* is often interchanged with the term *edge* or *link* in describing a relationship.

Unlike other solvers that PROC OPTMODEL uses, the network solver operates directly on arrays and sets. You do not need to explicitly define variables, constraints, and objectives to use the network solver. PROC OPTMODEL declares the appropriate objects internally as needed. You specify the names of arrays and sets that define your inputs and outputs as options in the SOLVE WITH NETWORK statement.

Getting Started: Network Solver

This section shows an introductory example for getting started with the network solver. For more information about the expected input formats and the various algorithms available, see the sections "Details: Network Solver" on page 398 and "Examples: Network Solver" on page 462.

Consider the following road network between a SAS employee's home in Raleigh, NC, and the SAS headquarters in Cary, NC.

In this road network (graph), the links are the roads and the nodes are intersections between roads. For each road, you assign a *link attribute* in the parameter time_to_travel to describe the number of minutes that it takes to drive from one node to another. The following data were collected using Google Maps (Google 2011):

```
data LinkSetInRoadNC10am;
   input start_inter $1-20 end_inter $20-40 miles miles_per_hour;
   datalines;
614CapitalBlvd      Capital/WadeAve      0.6  25
614CapitalBlvd      Capital/US70W        0.6  25
614CapitalBlvd      Capital/US440W       3.0  45
Capital/WadeAve     WadeAve/RaleighExpy  3.0  40
Capital/US70W       US70W/US440W         3.2  60
US70W/US440W        US440W/RaleighExpy   2.7  60
Capital/US440W      US440W/RaleighExpy   6.7  60
US440W/RaleighExpy  RaleighExpy/US40W    3.0  60
WadeAve/RaleighExpy RaleighExpy/US40W    3.0  60
RaleighExpy/US40W   US40W/HarrisonAve    1.3  55
US40W/HarrisonAve   SASCampusDrive       0.5  25
;
```

Using the network solver, you want to find the route that yields the shortest path between home (614 Capital Blvd) and the SAS headquarters (SAS Campus Drive). This can be done by using the SHORTPATH= option as follows:

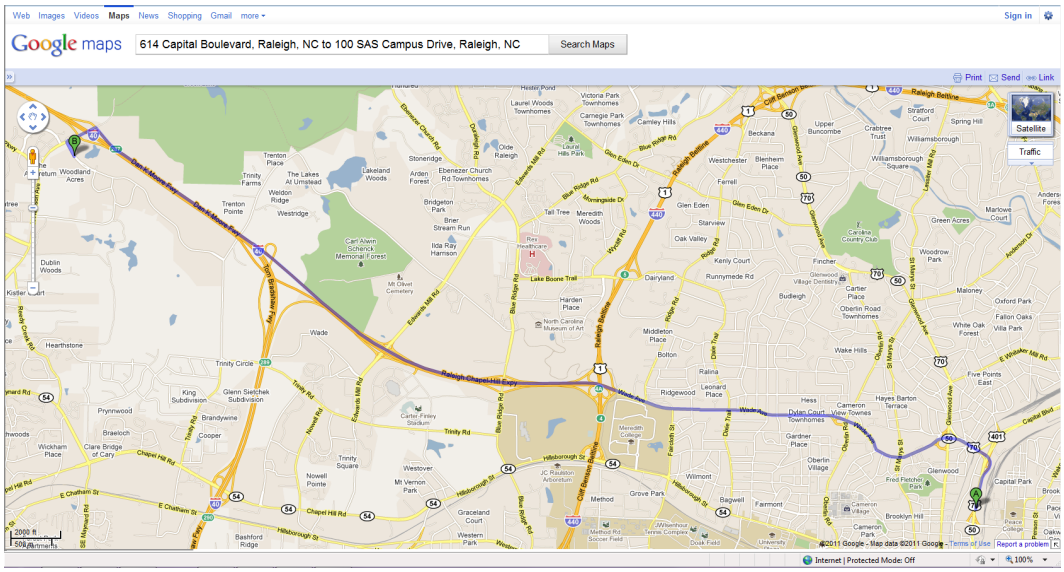
```
proc optmodel;
   set<str,str> LINKS;
   num miles{LINKS};
   num miles_per_hour{LINKS};
   num time_to_travel{<i,j> in LINKS} = miles[i,j]/ miles_per_hour[i,j] * 60;
   read data LinkSetInRoadNC10am into
      LINKS=[start_inter end_inter]
      miles miles_per_hour
   ;
   /* You can compute paths between many pairs of source and destination,
      so these parameters are declared as sets */
   set HOME = /"614CapitalBlvd"/;
   set WORK = /"SASCampusDrive"/;

   /* The path is stored as a set of: Start, End, Sequence, Tail, Head */
   set<str,str,num,str,str> PATH;

   solve with network /
      links     = ( weight = time_to_travel )
      shortpath = ( source = HOME
                    sink   = WORK )
      out       = ( sppaths = PATH )
   ;
   create data ShortPath from [s t order start_inter end_inter]=PATH
      time_to_travel[start_inter,end_inter];
quit;
```

For more information about shortest path algorithms in the network solver, see the section "Shortest Path" on page 436. Figure 9.1 displays the output data set ShortPath, which shows the best route to take to minimize travel time at 10:00 a.m. This route is also shown in Google Maps in Figure 9.2.

Figure 9.1 Shortest Path for Road Network at 10:00 A.M.

order	start_inter	end_inter	time_to_travel
1	614CapitalBlvd	Capital/WadeAve	1.4400
2	Capital/WadeAve	WadeAve/RaleighExpy	4.5000
3	WadeAve/RaleighExpy	RaleighExpy/US40W	3.0000
4	RaleighExpy/US40W	US40W/HarrisonAve	1.4182
5	US40W/HarrisonAve	SASCampusDrive	1.2000
			11.5582

Figure 9.2 Shortest Path for Road Network at 10:00 A.M. in Google Maps



Now suppose that it is rush hour (5:00 p.m.) and the time to traverse the roads has changed because of traffic patterns. You want to find the route that is the shortest path for going home from SAS headquarters under different speed assumptions due to traffic.

The following statements are similar to the first network solver run, except that one *miles_per_hour* value is modified and the SOURCE= and SINK= option values are reversed:

```
proc optmodel;
   set<str,str> LINKS;
   num miles{LINKS};
   num miles_per_hour{LINKS};
   num time_to_travel{<i,j> in LINKS} = miles[i,j]/ miles_per_hour[i,j] * 60;
   read data LinkSetInRoadNC10am into
      LINKS=[start_inter end_inter]
      miles miles_per_hour
   ;
   /* high traffic */
   miles_per_hour['Capital/WadeAve','WadeAve/RaleighExpy'] = 25;

   /* You can compute paths between many pairs of source and destination,
      so these parameters are declared as sets */
   set HOME = /"614CapitalBlvd"/;
   set WORK = /"SASCampusDrive"/;

   /* The path is stored as a set of: Start, End, Sequence, Tail, Head */
   set<str,str,num,str,str> PATH;

   solve with network /
      links     = ( weight = time_to_travel )
      shortpath = ( source = WORK
                    sink   = HOME )
      out       = ( sppaths = PATH )
   ;
   create data ShortPath from [s t order start_inter end_inter]=PATH
      time_to_travel[start_inter,end_inter];
quit;
```

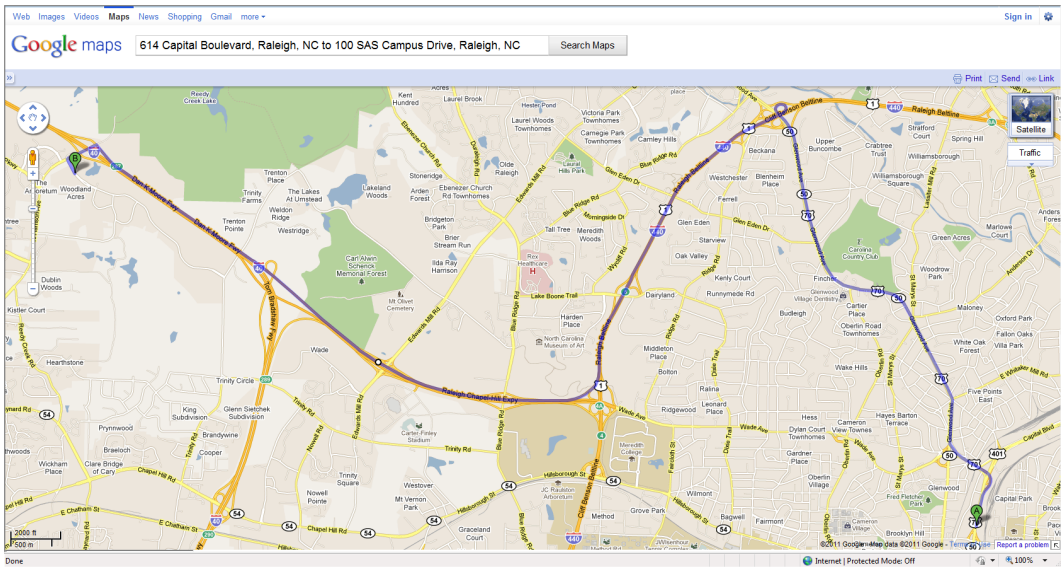
Now, the output data set ShortPath, shown in Figure 9.3, shows the best route for going home at 5:00 p.m. Because the traffic on Wade Avenue is usually heavy at this time of day, the route home is different from the route to work.

Figure 9.3 Shortest Path for Road Network at 5:00 P.M.

order	start_inter	end_inter	time_to_travel
1	US40W/HarrisonAve	SASCampusDrive	1.2000
2	RaleighExpy/US40W	US40W/HarrisonAve	1.4182
3	US440W/RaleighExpy	RaleighExpy/US40W	3.0000
4	US70W/US440W	US440W/RaleighExpy	2.7000
5	Capital/US70W	US70W/US440W	3.2000
6	614CapitalBlvd	Capital/US70W	1.4400
			12.9582

This new route is shown in Google Maps in Figure 9.4.

Figure 9.4 Shortest Path for Road Network at 5:00 P.M. in Google Maps



Syntax: Network Solver

SOLVE WITH NETWORK /

General and Diagnostic Options:
< **GRAPH_DIRECTION=DIRECTED | UNDIRECTED** >
< **INCLUDE_SELFLINK** >
< **LOGFREQ=***number* >
< **LOGLEVEL=***number* | *string* >
< **MAXTIME=***number* >
< **TIMETYPE=***number* | *string* >

Data Input and Output Options:
< **LINKS=(** *suboptions* **)** >
< **NODES=(** *suboptions* **)** >
< **OUT=(** *suboptions* **)** >
< **SUBGRAPH=(** *suboptions* **)** >

Algorithm Options:
< **BICONCOMP**< **=()** > >
< **CLIQUE**< **=(** *suboption* **)** > >
< **CONCOMP**< **=(** *suboption* **)** > >
< **CYCLE**< **=(** *suboptions* **)** > >
< **LINEAR_ASSIGNMENT**< **=()** > >
< **MINCOSTFLOW**< **=()** > >
< **MINCUT**< **=(** *suboptions* **)** > >
< **MINSPANTREE**< **=()** > >
< **SHORTPATH**< **=(** *suboptions* **)** > >
< **TRANSITIVE_CLOSURE**< **=()** > >
< **TSP**< **=(** *suboptions* **)** > > **;**

There are three types of SOLVE WITH NETWORK statement options:

- General and diagnostic options have the same meaning for multiple algorithms.
- Data input and output options, such as the LINKS=, NODES=, and OUT= options, control the names of the sets and variables that the network solver uses to build the graph and that the algorithms use for output.
- Algorithm options select an algorithm to run, and where available, provide further algorithm-specific configuration directives.

The section "Functional Summary" on page 382 provides a quick reference for each of the suboptions for each option. Each option is then described in more detail in its own section, in alphabetical order.

Functional Summary

Table 9.2 summarizes the options and suboptions available in the SOLVE WITH NETWORK statement.

Table 9.2 Functional Summary of SOLVE WITH NETWORK Options

Description	Option Suboption
General Options	
Specifies directed or undirected graphs	GRAPH_DIRECTION=
Includes self links in the graph definition	INCLUDE_SELFLINK=
Specifies the iteration log frequency	LOGFREQ=
Controls the amount of information that is displayed in the SAS log	LOGLEVEL=
Specifies the maximum time spent calculating results	MAXTIME=
Specifies whether time units are in CPU time or real time	TIMETYPE=
Input and Output Options	
Groups link-indexed data	LINKS=()
Names a set of links to include in the graph definition even if no weights or bounds are available for them	INCLUDE=
Specifies the flow lower bound for each link	LOWER=
Specifies the flow upper bound for each link	UPPER=
Specifies link weights	WEIGHT=
Groups node-indexed data	NODES=()
Names a set of nodes to include in the graph definition even if no weights are available for them	INCLUDE=
Specifies node weights	WEIGHT=
Specifies node supply upper bounds in the minimum-cost network flow problem	WEIGHT2=
Specifies the input sets that enable you to solve a problem over a subgraph	SUBGRAPH=()
Specifies the subset of links to use	LINKS=
Specifies the subset of nodes to use	NODES=
Specifies the output sets or arrays for each algorithm (see Table 9.4 for which OUT= suboptions you can specify for each algorithm)	OUT=()
Specifies the output set for articulation points	ARTPOINTS=
Specifies the output set for linear assignment	ASSIGNMENTS=
Specifies the array to contain the biconnected component of each link	BICONCOMP=
Specifies the output set for cliques	CLIQUES=
Specifies the output array for connected components	CONCOMP=
Specifies the output set for the cut-sets for minimum cuts	CUTSETS=
Specifies the output set for cycles	CYCLES=
Specifies the output array for the flow on each link	FLOW=
Specifies the output set for the minimum spanning tree (forest)	FOREST=
Specifies the output set for the links that remain after the SUBGRAPH= option is applied	LINKS=

Table 9.2 *(continued)*

Description	Option Suboption
Specifies the output set for the nodes that remain after the SUBGRAPH= option is applied	NODES=
Specifies the output array for the node order in the traveling salesman problem	ORDER=
Specifies the output set for the partitions for minimum cuts	PARTITIONS=
Specifies the set to contain the link sequence for each path	SPPATHS=
Specifies the numeric array to contain the path weight for each source and sink node pair	SPWEIGHTS=
Specifies the output set for the tour in the traveling salesman problem	TOUR=
Specifies the set to contain the pairs (u, v) of nodes where v is reachable from u	TRANSCL=
Algorithm Options and Suboptions	
Finds biconnected components and articulation points of an undirected input graph	BICONCOMP
Finds maximal cliques in the input graph	CLIQUE=
Specifies the maximum number of cliques to return	MAXCLIQUES=
Finds the connected components of the input graph	CONCOMP=
Specifies the algorithm to use for calculating connected components	ALGORITHM=
Finds the cycles (or the existence of a cycle) in the input graph	CYCLE=
Specifies the maximum number of cycles to return	MAXCYCLES=
Specifies the maximum link count for the cycles to return	MAXLENGTH=
Specifies the maximum link weight for the cycles to return	MAXLINKWEIGHT=
Specifies the maximum sum of node weights to allow in a cycle	MAXNODEWEIGHT=
Specifies the minimum link count for the cycles to return	MINLENGTH=
Specifies the minimum link weight for the cycles to return	MINLINKWEIGHT=
Specifies the minimum node weight for the cycles to return	MINNODEWEIGHT=
Specifies whether to stop after finding the first cycle	MODE=
Solves the minimal-cost linear assignment problem	LINEAR_ASSIGNMENT
Solves the minimum-cost network flow problem	MINCOSTFLOW
Finds the minimum link-weighted cut of an input graph	MINCUT=
Specifies the maximum number of cuts to return from the algorithm	MAXNUMCUTS=
Specifies the maximum weight of each cut to return from the algorithm	MAXWEIGHT=
Solves the minimum link-weighted spanning tree problem on an input graph	MINSPANTREE
Calculates shortest paths between sets of nodes on the input graph	SHORTPATH=
Specifies the type of output for shortest paths results	PATHS=
Specifies the set of sink nodes	SINK=
Specifies the set of source nodes	SOURCE=
Specifies whether to use weights in calculating shortest paths	USEWEIGHT=
Calculates the transitive closure of an input graph	TRANSITIVE_CLOSURE

Table 9.2 *(continued)*

Description	Option Suboption
Solves the traveling salesman problem	TSP=
Requests that the stopping criterion be based on the absolute objective gap	ABSOBJGAP=
Specifies the level of conflict search	CONFLICTSEARCH=
Specifies the cutoff value for branch-and-bound node removal	CUTOFF=
Specifies the level of cutting planes to be generated by the network solver	CUTSTRATEGY=
Emphasizes feasibility or optimality	EMPHASIS=
Specifies the initial and primal heuristics level	HEURISTICS=
Specifies the maximum number of branch-and-bound nodes to be processed	MAXNODES=
Specifies the maximum number of feasible tours to be identified	MAXSOLS=
Specifies whether to use a mixed integer linear programming solver	MILP=
Specifies the branch-and-bound node selection strategy	NODESEL=
Specifies the probing level	PROBE=
Requests that the stopping criterion be based on relative objective gap	RELOBJGAP=
Specifies the number of simplex iterations to be performed on each variable in the strong branching strategy	STRONGITER=
Specifies the number of candidates for the strong branching strategy	STRONGLEN=
Requests that the stopping criterion be based on the target objective value	TARGET=
Specifies the rule for selecting branching variable	VARSEL=

Table 9.3 lists the valid GRAPH_DIRECTION= values for each algorithm option in the SOLVE WITH NETWORK statement.

Table 9.3 Supported Graph Directions by Algorithm

	Direction	
Algorithm	Undirected	Directed
BICONCOMP	x	
CLIQUE	x	
CONCOMP	x	x
CYCLE	x	x
LINEAR_ASSIGNMENT		x
MINCOSTFLOW		x
MINCUT	x	
MINSPANTREE	x	
SHORTPATH	x	x
TRANSITIVE_CLOSURE	x	x
TSP	x	x

Table 9.4 indicates, for each algorithm option in the SOLVE WITH NETWORK statement, which output options you can specify, and what their types can be. The types vary depending on whether nodes are of type STRING or NUMBER.

Table 9.4 Output Suboptions and Types by Algorithm

Algorithm Option **OUT= Suboption**	**OPTMODEL Type**
BICONCOMP	
ARTPOINTS=	SET<STRING> or SET<NUMBER>
BICONCOMP=	NUMBER indexed over links (<NUMBER,NUMBER> or <STRING,STRING>)
CLIQUE	
CLIQUES=	SET<NUMBER,NUMBER> or SET<NUMBER,STRING>
CONCOMP	
CONCOMP=	NUMBER indexed over nodes (NUMBER or STRING)
CYCLE	
CYCLES=	SET<NUMBER,NUMBER,NUMBER> or SET<NUMBER,NUMBER,STRING>
LINEAR_ASSIGNMENT	
ASSIGNMENTS=	SET<NUMBER,NUMBER> or SET<STRING,STRING>
MINCOSTFLOW	
FLOW=	NUMBER indexed over links (<NUMBER,NUMBER> or <STRING,STRING>)
MINCUT	
CUTSETS=	SET<NUMBER,NUMBER,NUMBER> or SET<NUMBER,STRING,STRING>
PARTITIONS=	SET<NUMBER,NUMBER> or SET<NUMBER,STRING>
MINSPANTREE	
FOREST=	SET<NUMBER,NUMBER> or SET<STRING,STRING>
SHORTPATH	
SPPATHS=	SET<NUMBER,NUMBER,NUMBER,NUMBER,NUMBER> or SET<STRING,STRING,NUMBER,STRING,STRING>
SPWEIGHTS=	NUMBER indexed over sink and source node pairs (<NUMBER,NUMBER> or <STRING,STRING>)
TRANSITIVE_CLOSURE	
CLOSURE=	SET<NUMBER,NUMBER> or SET<STRING,STRING>
TSP	
ORDER=	NUMBER indexed over nodes (NUMBER or STRING)
TOUR=	SET<NUMBER,NUMBER> or SET<STRING,STRING>

SOLVE WITH NETWORK Statement

SOLVE WITH NETWORK */ < options >* **;**

The SOLVE WITH NETWORK statement invokes the network solver. You can specify the following *options* to define various processing and diagnostic controls, the graph input and output, and the algorithm to run:

General Options

You can specify the following general options, which have the same meaning for multiple algorithms.

GRAPH_DIRECTION=DIRECTED | UNDIRECTED
DIRECTION=DIRECTED | UNDIRECTED
specifies directed or undirected graphs.

Table 9.5 Values for the GRAPH_DIRECTION= Option

Option Value	Description
DIRECTED	Requests a directed graph. In a directed graph, each link (i, j) has a direction that defines how something (for example, information) might flow over that link. In link (i, j), information flows from node i to node j ($i \rightarrow j$). The node i is called the *source* (*tail*) node, and node j is called the *sink* (*head*) node.
UNDIRECTED	Requests an undirected graph. In an undirected graph, each link $\{i, j\}$ has no direction and information can flow in either direction. That is, $\{i, j\} = \{j, i\}$.

By default, GRAPH_DIRECTION=UNDIRECTED.

INCLUDE_SELFLINK
includes self links in the graph definition—for example, (i, i)—when an input graph is read. By default, when the network solver reads the LINKS= data, it removes all self links.

LOGFREQ=*number*
controls the frequency with which an algorithm reports progress from its underlying solver. This setting is recognized by the traveling salesman problem and minimum-cost flow algorithms. You can set *number* to 0 to turn off log updates from underlying algorithms.

LOGLEVEL=*number* **|** *string*
controls the amount of information that is displayed in the SAS log. This setting sets the log level for all algorithms. Table 9.6 describes the valid values for this option.

Table 9.6 Values for LOGLEVEL= Option

number	*string*	Description
0	NONE	Turns off all procedure-related messages in the SAS log

Table 9.6 *(continued)*

number	*string*	**Description**
1	BASIC	Displays a basic summary of the input, output, and algorithmic processing
2	MODERATE	Displays a summary of the input, output, and algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the input, output, and algorithmic processing

By default, LOGLEVEL=BASIC.

MAXTIME=*number*

specifies the maximum time spent calculating results. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment. The clique, cycle, minimum-cost network flow, and traveling salesman problem algorithms recognize the MAXTIME= option.

TIMETYPE=*number* | *string*

specifies whether CPU time or real time is used for the MAXTIME= option for each applicable algorithm. Table 9.7 describes the valid values of the TIMETYPE= option.

Table 9.7 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

By default, TIMETYPE=CPU.

Input and Output Options

The following options enable you to specify the graph to run algorithms on. These options take array and set names. They are known as *identifier expressions* in Chapter 5, "The OPTMODEL Procedure." Also see Table 9.4 for semantic requirements and the section "Input Data for the Network Solver" on page 398 for use cases.

LINKS=(*suboptions* **)**

groups link-indexed data. For more information, see the section "Input Data for the Network Solver" on page 398.

You can specify the following *suboptions*:

INCLUDE=*set-name*

names a set of links to include in the graph definition even if no weights or bounds are available for them. For more information, see "Example 9.1: Articulation Points in a Terrorist Network" on page 462. The set must be numeric, and it must be indexed over a subset of the links of the graph.

LOWER=*array-name*
specifies the flow lower bound for each link. The array must be numeric, and it must be indexed over a subset of the inks of the graph.

UPPER=*array-name*
specifies the flow upper bound for each link. The array must be numeric, and it must be indexed over a subset of the links of the graph.

WEIGHT=*array-name*
specifies link weights. The array must be numeric, and it must be indexed over a subset of the links of the graph. If you specify this suboption, then any link that does not appear in the index set of the WEIGHT= array has weight 0. If you do not specify this suboption, then every link has weight 1.

NODES=(*suboptions* **)**
groups node-indexed data. For more information, see the section "Input Data for the Network Solver" on page 398.

You can specify the following *suboptions*:

INCLUDE=*set-name*
names a set of nodes to include in the graph definition even if no weights are available for them. For more information, see the section "Connected Components" on page 413.

WEIGHT=*array-name*
specifies node weights. The array must be numeric, and it must be indexed over a subset of the nodes of the graph.

WEIGHT2=*array-name*
specifies node supply upper bounds in the minimum-cost network flow problem. The array must be numeric, and it must be indexed over a subset of the nodes of the graph. For more information, see the section "Minimum-Cost Network Flow" on page 424.

OUT=(*suboptions* **)**
specifies the output sets or arrays for each algorithm (see Table 9.4 for which OUT= suboptions you can specify for each algorithm). You can use some of these options (even if you do not invoke any algorithm) to see the filtering outcome that is produced by the SUBGRAPH= option.

If you do not specify a *suboption* that matches the algorithm option in the statement, the algorithm runs and only updates the objective.

If you specify a suboption that does not match the algorithm option in the statement, OPTMODEL issues a warning.

When you declare arrays that are indexed over nodes, over links, or over sets of nodes or links, you must use the same type you used in your node definition.

See the various algorithm sections for examples of the use of these OUT= *suboptions*.

ARTPOINTS=*set-name*
specifies the output set for articulation points. Each element of the set represents a node ID. This suboption is used with the BICONCOMP algorithm option.

ASSIGNMENTS=*set-name*
specifies the output set for linear assignment. This suboption is used with the LINEAR_ASSIGNMENT algorithm option.

BICONCOMP=*array-name*
specifies the array to contain the biconnected component of each link. This suboption is used with the BICONCOMP algorithm option.

CLIQUES=*set-name*
specifies the output set for cliques. Each tuple of the set represents clique ID and node ID. This suboption is used with the CLIQUE algorithm option.

CONCOMP=*array-name*
specifies the output array for connected components. This suboption is used with the CONCOMP algorithm option.

CUTSETS=*set-name*
specifies the output set for the cut-sets for minimum cuts. Each tuple of the set represents the cut ID, the tail node ID, and the head node ID. This suboption is used with the MINCUT algorithm option.

CYCLES=*set-name*
specifies the output set for cycles. Each tuple of the set represents a cycle ID, the order within that cycle, and the node ID. This suboption is used with the CYCLE algorithm option.

FLOW=*array-name*
specifies the output array for the flow on each link. This suboption is used with the MINCOSTFLOW algorithm option.

FOREST=*set-name*
specifies the output set for the minimum spanning tree (forest). This suboption is used with the MINSPANTREE algorithm option.

LINKS=*set-name*
specifies the output set for the links that remain after the SUBGRAPH= option is applied. Each tuple of the set represents tail and head nodes, followed by a sequence of numbers which correspond to the attributes you provide in the LINKS= suboptions. The length of the tuples must be the number of attributes you specify plus two (for the tail and head node information). The options you specify in the LINKS= option will appear in the output in the order: WEIGHT=, LOWER=, and UPPER=. For an example, see Figure 9.12 in "Solving over Subsets of Nodes and Links (Filters)" on page 402.

NODES=*set-name*
specifies the output set for the nodes that remain after the SUBGRAPH= option is applied. Each tuple of the set represents a node, followed by a sequence of numbers which correspond to the attributes you provide in the NODES= suboptions. The length of the tuples must be the number of attributes you specify plus one (for node information). The options you specify in the NODES= option will appear in the output in the order: WEIGHT= and WEIGHT2=. For an example, see the section "Minimum-Cost Network Flow with Flexible Supply and Demand" on page 428.

ORDER=*array-name*
specifies the numeric array to contain the position of each node within the optimal tour. This suboption is used with the TSP algorithm option.

PARTITIONS=*set-name*
specifies the output set for the partitions for minimum cuts. The set contains, for each partition, the node IDs in the smaller of the two subsets. Each tuple of the set represents a cut ID and a node ID. This suboption is used with the MINCUT algorithm option.

SPPATHS=*set-name*
specifies the set to contain the link sequence for each path. Each tuple of the set represents a source node ID, a sink node ID, a sequence number, a tail node ID, and a head node ID. This suboption is used with the SHORTPATH algorithm option.

SPWEIGHTS=*array-name*
specifies the numeric array to contain the path weight for each source and sink node pair. This suboption is used with the SHORTPATH algorithm option.

TOUR=*set-name*
specifies the output set for the tour in the traveling salesman problem. This suboption is used with the TSP algorithm option.

TRANSCL=*set-name*
specifies the set to contain the pairs (u, v) of nodes where v is reachable from u. This suboption is used with the TRANSITIVE_CLOSURE algorithm option.

SUBGRAPH=*(suboptions)*
specifies the input sets that enable you to solve a problem over a subgraph. For more information, see the section "Input Data for the Network Solver" on page 398.

You can specify the following *suboptions*:

LINKS=*set-name*
specifies the subset of links to use. If you specify a node pair that is not referenced in any of the suboptions of the LINKS= option, then the network solver returns an error.

NODES=*set-name*
specifies the subset of nodes to use. If you specify a node that is not referenced in any of the suboptions of the LINKS= option or the NODES= option, then the network solver returns an error.

Algorithm Options

BICONCOMP< =() >
finds biconnected components and articulation points of an undirected input graph. For more information, see the section "Biconnected Components and Articulation Points" on page 407.

CLIQUE< =(*suboption* **) >**
finds maximal cliques in the input graph. For more information, see the section "Clique" on page 410.

You can specify the following *suboption*:

MAXCLIQUES=*number*
specifies the maximum number of cliques to return. The default is the positive number that has the largest absolute value that can be represented in your operating environment.

CONCOMP< =(*suboption* **) >**
finds the connected components of the input graph. For more information, see the section "Connected Components" on page 413.

You can specify the following *suboption*:

ALGORITHM=DFS | UNION_FIND
specifies the algorithm to use for calculating connected components. Table 9.8 describes the valid values for this option.

Table 9.8 Values for the ALGORITHM= Option

Option Value	Description
DFS	Uses the depth-first search algorithm for connected components.
UNION_FIND	Uses the union-find algorithm for connected components. You can use ALGORITHM=UNION_FIND only with undirected graphs.

By default, ALGORITHM=UNION_FIND for undirected graphs, and ALGORITHM=DFS for directed graphs.

CYCLE< =(*suboptions* **) >**
finds the cycles (or the existence of a cycle) in the input graph. For more information, see the section "Cycle" on page 418.

You can specify the following *suboptions* in the CYCLE= option:

MAXCYCLES=*number*
specifies the maximum number of cycles to return. The default is the positive number that has the largest absolute value that can be represented in your operating environment. This option works only when you also specify MODE=ALL_CYCLES.

MAXLENGTH=*number*
specifies the maximum number of links to allow in a cycle. Any cycle whose length is greater than *number* is removed from the results. The default is the positive number that has the largest absolute value that can be represented in your operating environment. By default, nothing is removed from the results. This option works only when you also specify MODE=ALL_CYCLES.

MAXLINKWEIGHT=*number*
specifies the maximum sum of link weights to allow in a cycle. Any cycle whose sum of link weights is greater than *number* is removed from the results. The default is the positive number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

MAXNODEWEIGHT=*number*
specifies the maximum sum of node weights to allow in a cycle. Any cycle whose sum of node weights is greater than *number* is removed from the results. The default is the positive number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

MINLENGTH=*number*
specifies the minimum number of links to allow in a cycle. Any cycle that has fewer links than *number* is removed from the results. The default is 1. By default, only self-loops are filtered. This option works only when you also specify MODE=ALL_CYCLES.

MINLINKWEIGHT=*number*
specifies the minimum sum of link weights to allow in a cycle. Any cycle whose sum of link weights is less than *number* is removed from the results. The default the negative number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

MINNODEWEIGHT=*number*
specifies the minimum sum of node weights to allow in a cycle. Any cycle whose sum of node weights is less than *number* is removed from the results. The default is the negative number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

MODE=ALL_CYCLES | FIRST_CYCLE
specifies whether to stop after finding the first cycle. Table 9.9 describes the valid values for this option.

Table 9.9 Values for the MODE= Option

Option Value	Description
ALL_CYCLES	Returns all (unique, elementary) cycles found.
FIRST_CYCLE	Returns the first cycle found.

By default, MODE=FIRST_CYCLE.

LINEAR_ASSIGNMENT< =() >
LAP< =() >
solves the minimal-cost linear assignment problem. In graph terms, this problem is also known as the minimum link-weighted matching problem on a bipartite directed graph. The input data (the cost matrix) is defined as a directed graph by specifying the LINKS= option in the SOLVE WITH NETWORK statement, where the costs are defined as link weights. Internally, the graph is treated as a bipartite directed graph.

For more information, see the section "Linear Assignment (Matching)" on page 423.

MINCOSTFLOW< =() >

MCF< =() >

solves the minimum-cost network flow problem.

For more information, see the section "Minimum-Cost Network Flow" on page 424.

MINCUT< =(*suboptions* **) >**

finds the minimum link-weighted cut of an input graph. For more information, see the section "Minimum Cut" on page 430. You can specify the following *suboptions* in the MINCUT= option:

MAXNUMCUTS=*number*

specifies the maximum number of cuts to return from the algorithm. The minimal cut and any others found during the search, up to *number*, are returned. By default, MAXNUMCUTS=1.

MAXWEIGHT=*number*

specifies the maximum weight of the cuts to return from the algorithm. Only cuts that have weight less than or equal to *number* are returned. The default is the positive number that has the largest absolute value that can be represented in your operating environment.

MINSPANTREE< =() >

MST< =() >

solves the minimum link-weighted spanning tree problem on an input graph. For more information, see the section "Minimum Spanning Tree" on page 434.

SHORTPATH< =(*suboptions* **) >**

calculates shortest paths between sets of nodes on the input graph. For more information, see the section "Shortest Path" on page 436.

You can specify the following suboptions:

PATHS=ALL | LONGEST | SHORTEST

specifies the type of output for shortest paths results.

Table 9.10 lists the valid values for this suboption.

Table 9.10 Values for the PATHS= Option

Option Value	Description
ALL	Outputs shortest paths for all pairs of source-sinks.
LONGEST	Outputs shortest paths for the source-sink pair that has the longest (finite) length. If other source-sink pairs (up to 100) have equally long length, they are also output.
SHORTEST	Outputs shortest paths for the source-sink pair that has the shortest length. If other source-sink pairs (up to 100) have equally short length, they are also output.

By default, SHORTPATH=ALL.

SINK=*set-name*
specifies the set of sink nodes.

SOURCE=*set-name*
specifies the set of source nodes.

USEWEIGHT=YES | NO
specifies whether to use weights in calculating shortest paths as listed in Table 9.11.

Table 9.11 Values for USEWEIGHT= Option

Option Value	Description
YES	Uses weights (if they exist) in shortest path calculations.
NO	Does not use weights in shortest path calculations.

By default, USEWEIGHT=YES.

TRANSITIVE_CLOSURE< =() >
TRANSCL< =() >
calculates the transitive closure of an input graph. For more information, see the section "Transitive Closure" on page 449.

TSP< =(*suboptions* **) >**
solves the traveling salesman problem. For more information, see the section "Traveling Salesman Problem" on page 452.

The algorithm that is used to solve this problem is built around the same method as is used in PROC OPTMILP: a branch-and-cut algorithm. Many of the following *suboptions* are the same as those described for the OPTMILP procedure in the *SAS/OR User's Guide: Mathematical Programming*.

You can specify the following *suboptions*:

ABSOBJGAP=*number*
specifies a stopping criterion. When the absolute difference between the best integer objective and the objective of the best remaining branch-and-bound node becomes less than the value of *number*, the solver stops. The value of *number* can be any nonnegative number. By default, ABSOBJGAP=1E–6.

CONFLICTSEARCH=*number* | *string*
specifies the level of conflict search that the network solver performs. The solver performs a conflict search to find clauses that result from infeasible subproblems that arise in the search tree. Table 9.12 describes the valid values for this option.

Table 9.12 Values for CONFLICTSEARCH= Option

number	*string*	Description
–1	AUTOMATIC	Performs a conflict search based on a strategy that is determined by the network solver

Table 9.12 *(continued)*

number	*string*	**Description**
0	NONE	Disables conflict search
1	MODERATE	Performs a moderate conflict search
2	AGGRESSIVE	Performs an aggressive conflict search

By default, CONFLICTSEARCH=AUTOMATIC.

CUTOFF=*number*

cuts off any branch-and-bound nodes in a minimization problem that has an objective value that is greater than *number*. The value of *number* can be any number.

The default value is the positive number that has the largest absolute value that can be represented in your operating environment.

CUTSTRATEGY=*number* | *string*

specifies the level of cutting planes to be generated by the network solver. TSP-specific cutting planes are always generated. Table 9.13 describes the valid values for this option.

Table 9.13 Values for CUTSTRATEGY= Option

number	*string*	**Description**
–1	AUTOMATIC	Generates cutting planes based on a strategy determined by the mixed integer linear programming solver
0	NONE	Disables generation of mixed integer programming cutting planes (some TSP-specific cutting planes are still active for validity)
1	MODERATE	Uses a moderate cut strategy
2	AGGRESSIVE	Uses an aggressive cut strategy

By default, CUTSTRATEGY=NONE.

EMPHASIS=*number* | *string*

specifies a search emphasis option. Table 9.14 describes the valid values for this option.

Table 9.14 Values for EMPHASIS= Option

number	*string*	**Description**
0	BALANCE	Performs a balanced search
1	OPTIMAL	Emphasizes optimality over feasibility
2	FEASIBLE	Emphasizes feasibility over optimality

By default, EMPHASIS=BALANCE.

HEURISTICS=*number* | *string*

controls the level of initial and primal heuristics that the network solver applies. This level determines how frequently the network solver applies primal heuristics during the branch-and-bound tree search. It also affects the maximum number of iterations that are allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. Table 9.15 lists the valid values for this option.

Table 9.15 Values for HEURISTICS= Option

number	*string*	**Description**
–1	AUTOMATIC	Applies the default level of heuristics
0	NONE	Disables all initial and primal heuristics
1	BASIC	Applies basic initial and primal heuristics at low frequency
2	MODERATE	Applies most initial and primal heuristics at moderate frequency
3	AGGRESSIVE	Applies all initial primal heuristics at high frequency

By default, HEURISTICS=AUTOMATIC.

MAXNODES=*number*

specifies the maximum number of branch-and-bound nodes to be processed. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$.

By default, MAXNODES=$2^{31} - 1$.

MAXSOLS=*number*

specifies the maximum number of feasible tours to be identified. If *number* solutions have been found, then the solver stops. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$.

By default, MAXSOLS=$2^{31} - 1$.

MILP=*number* | *string*

specifies whether to use a mixed integer linear programming (MILP) solver for solving the traveling salesman problem. The MILP solver attempts to find the overall best TSP tour by using a branch-and-bound based algorithm. This algorithm can be expensive for large-scale problems. If MILP=OFF, then the network solver uses its initial heuristics to find a feasible, but not necessarily optimal, tour as quickly as possible. Table 9.16 describes the valid values for this option.

Table 9.16 Values for MILP= Option

number	*string*	**Description**
1	ON	Uses a mixed integer linear programming solver
0	OFF	Does not use a mixed integer linear programming solver

By default, MILP=ON

NODESEL=*number* | *string*
specifies the branch-and-bound node selection strategy option. For more information about node selection, see Chapter 13, "The OPTMILP Procedure." Table 9.17 describes the valid values for this option.

Table 9.17 Values for NODESEL= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses automatic node selection
0	BESTBOUND	Chooses the node that has the best relaxed objective (best-bound-first strategy)
1	BESTESTIMATE	Chooses the node that has the best estimate of the integer objective value (best-estimate-first strategy)
2	DEPTH	Chooses the most recently created node (depth-first strategy)

By default, NODESEL=AUTOMATIC.

PROBE=*number* | *string*
specifies a probing option. Table 9.18 describes the valid values for this option.

Table 9.18 Values for PROBE= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses an automatic probing strategy
0	NONE	Disables probing
1	MODERATE	Uses the probing moderately
2	AGGRESSIVE	Uses the probing aggressively

By default, PROBE=NONE.

RELOBJGAP=*number*
specifies a stopping criterion that is based on the best integer objective (BestInteger) and the objective of the best remaining node (BestBound). The relative objective gap is equal to

$$| \text{BestInteger} - \text{BestBound} | / (\text{1E–10} + | \text{BestBound} |)$$

When this value becomes less than the specified gap size *number*, the solver stops. The value of *number* can be any nonnegative number.

By default, RELOBJGAP=1E–4.

STRONGITER=*number* | **AUTOMATIC**
specifies the number of simplex iterations that the network solver performs for each variable in the candidate list when it uses the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, the network solver uses the default value, which it calculates automatically.

STRONGLEN=*number* | **AUTOMATIC**
specifies the number of candidates that the network solver considers when it uses the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, the network solver uses the default value, which it calculates automatically.

TARGET=*number*
specifies a stopping criterion for minimization problems. If the best integer objective is better than or equal to *number*, the solver stops. The value of *number* can be any number.

By default, TARGET is the negative number that has the largest absolute value that can be represented in your operating environment.

VARSEL=*number* | *string*
specifies the rule for selecting the branching variable. For more information about variable selection, see Chapter 13, "The OPTMILP Procedure." Table 9.19 describes the valid values for this option.

Table 9.19 Values for VARSEL= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses automatic branching variable selection
0	MAXINFEAS	Chooses the variable that has maximum infeasibility
1	MININFEAS	Chooses the variable that has minimum infeasibility
2	PSEUDO	Chooses a branching variable based on pseudocost
3	STRONG	Uses the strong branching variable selection strategy

By default, VARSEL=AUTOMATIC.

Details: Network Solver

The network solver uses a collection of specialized algorithms that optimize specific types of common problems. When you use the network solver, you specify variable arrays, numeric arrays, and sets, both to define an instance and to get solutions, without explicitly formulating objectives and constraints.

Input Data for the Network Solver

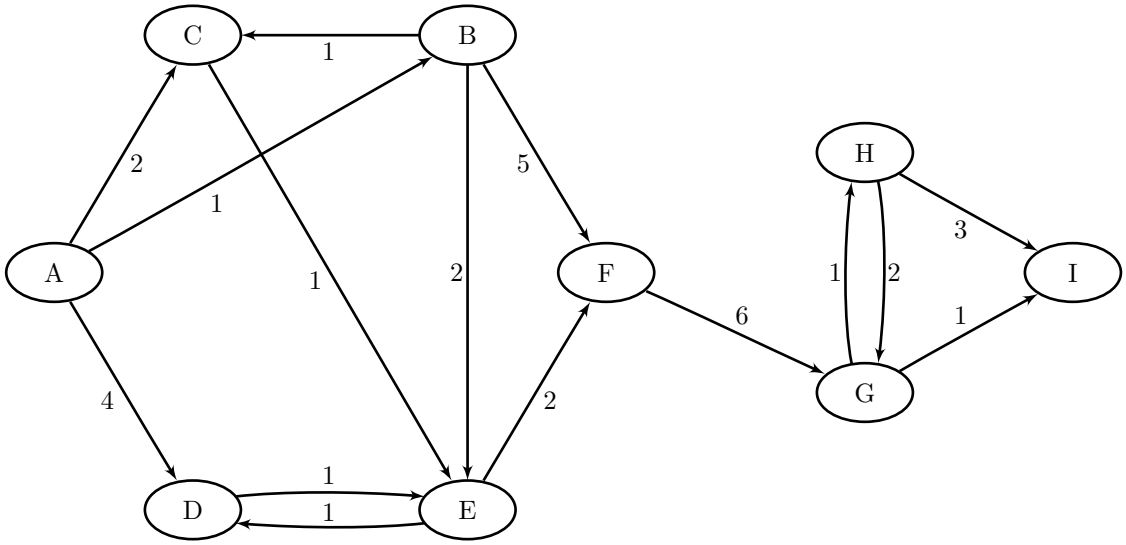
This section describes how you can import and export node, link, and problem data from and to SAS data sets and how you can solve problems over a subgraph without changing your original sets. The section "Graph Input Data" on page 399 describes how to load node and link data in some common formats. The section "Solving over Subsets of Nodes and Links (Filters)" on page 402 describes subgraphs and how to access the objective value of a network problem.

Graph Input Data

This section describes how to input a graph for analysis by the network solver. Because PROC OPTMODEL uses node and link attributes that are indexed over the sets of nodes and links, you need to provide only node and link attributes. PROC OPTMODEL infers the graph from the attributes you provide. When a documented default value exists for the attribute of a link or a node, you need to provide only the values that differ from the default. For example, the section "Minimum-Cost Network Flow" on page 424 assumes that the link flow upper bound is ∞. You need to specify only the finite upper bounds.

Consider the directed graph shown in Figure 9.5.

Figure 9.5 A Simple Directed Graph



Each node and link has associated attributes: a node label and a link weight.

None of the algorithms in PROC OPTMODEL support Null Graphs, i.e., graphs with 0 nodes. PROC OPTMODEL will usually raise a semantic error and stop processing any remaining statements in a block if after processing its inputs it determines that the graph is null. If the graph definition itself is not null, but the graph to be passed to the solver after applying the SUBGRAPH= option is null, then the predeclared parameter _SOLUTION_STATUS_ will be set to NULL_GRAPH. For more information, see "Solving over Subsets of Nodes and Links (Filters)" on page 402.

Data Indexed by Nodes or Links

Nodes often represent entities, and links represent relationships between these entities. Therefore, it is common to store a graph as a link-indexed table. When nodes have attributes beyond their name (label), these attributes are stored in a node-indexed table. This section covers the more complex link-indexed case. The node-indexed case is essentially identical to this one, except that the PROC OPTMODEL set has tuple length of one when node-indexed data are read, whereas the PROC OPTMODEL set has tuple length two when link-indexed data are read.

Let $G = (N, A)$ define a graph with a set N of nodes and a set A of links. A link is an ordered pair of nodes. Each node is defined by using either numeric or string labels.

The directed graph G shown in Figure 9.5 can be represented by the following links data set LinkSetIn:

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 1   A C 2   A D 4   B C 1   B E 2
B F 5   C E 1   D E 1   E D 1   E F 2
F G 6   G H 1   G I 1   H G 2   H I 3
;
```

The following statements read in this graph and print the resulting links and nodes sets. These statements do not run any algorithms, so the resulting output contains only the input graph.

```
proc optmodel;
   set<str,str> LINKS;
   set NODES = union{<ni,nj> in LINKS} {ni,nj};
   num weight{LINKS};

   read data LinkSetIn into LINKS=[from to] weight;

   print weight;
   put NODES=; /* computed automatically by OPTMODEL */
quit;
```

The network solver preserves the node order of each link that you provide, even in cases where the link is traversed in the opposite order, such as in paths or tours. For an example, see the tour $(1, 4, 2, 3, 5)$ in Figure 9.8.

The log output in Figure 9.6 shows the nodes that are read from the input link data set. In this example PROC OPTMODEL computed the node set N (NODES) from its definition when it was needed. The ODS output in Figure 9.7 shows the weights that are read from the input link data set, which is indexed by link. PUT is used for NODES because PROC OPTMODEL sets are basic types such as number and string. Thus, you use PUT to quickly inspect a set value. In contrast, you use PRINT to inspect an array, such as weight.

Figure 9.6 Node Set Printout of a Simple Directed Graph

```
NOTE: There were 15 observations read from the data set WORK.LINKSETIN.
NODES={'A','B','C','D','E','F','G','H','I'}
```

Figure 9.7 Link Set of a Simple Directed Graph That Includes Weights

The OPTMODEL Procedure

[1]	[2]	weight
A	B	1
A	C	2
A	D	4
B	C	1
B	E	2
B	F	5
C	E	1
D	E	1
E	D	1
E	F	2
F	G	6
G	H	1
G	I	1
H	G	2
H	I	3

As described in the GRAPH_DIRECTION= option, if the graph is undirected, the *from* and *to* labels are interchangeable. If you define this graph as undirected, then reciprocal links (for example, $D \rightarrow E$ and $E \rightarrow D$) are treated as the same link, and duplicates are removed. The network solver takes the first occurrence of the link and ignores the others. By default, GRAPH_DIRECTION=UNDIRECTED, so to declare the graph as undirected you can just omit this option.

After you read the data into PROC OPTMODEL sets, you pass link information to the solver by using the LINKS= option. Node input is analogous to link input. You pass node information to the solver by using the NODES= option.

The INCLUDE= suboption is especially useful for algorithms that depend only on the graph topology, (such as the connected components algorithm). If an algorithm requires a node or link property and that property is not defined for a node or link that is added by the INCLUDE= suboption, the algorithm will not run.

Matrix Input Data

The contents of a table can be represented as a graph. The relationships between two sets of nodes, N_1 and N_2, can be represented by a $|N_1|$ by $|N_2|$ incidence matrix A, in which N_1 is the set of rows and N_2 is the set of columns.

To read a matrix that is stored in a data set into PROC OPTMODEL, you need to take two extra steps:

1. Determine the name of each numeric variable that you want to use. PROC CONTENTS can be useful for this task.
2. Use an iterated READ DATA statement.

For more information, see "Example 9.3: Linear Assignment Problem for Minimizing Swim Times" on page 468.

Solving over Subsets of Nodes and Links (Filters)

You can solve a problem over a subgraph without declaring new link and node sets. You can specify the LINKS= and NODES= suboptions of the SUBGRAPH= option to filter nodes and links before PROC OPTMODEL builds and solves the instance. If you want to see the resulting subgraph, you can specify the LINKS= and NODES= suboptions of the OUT= option. If you just want to produce a subgraph, you do not need to invoke an algorithm.

You can keep all the input and output arrays defined over the original graph and define a subgraph by providing any combination of the LINKS= and NODES= suboptions of the SUBGRAPH= option. If you specify either of the suboptions of the SUBGRAPH= option, then union semantics apply. PROC OPTMODEL uses the following rules:

- Only the links that are included in the set named in the LINKS= option are used to create the instance.
- Only the nodes that appear either in the NODES= suboption of the SUBGRAPH= option or that appear as the head or tail of a link in the LINKS= suboption are used to create the instance.
- A node or a link that appears only in the SUBGRAPH= option, but not in the original graph, is discarded. To add nodes or links that do not have attributes, see the INCLUDE= suboption of the LINKS= and NODE= options.

If the value of the LOGLEVEL= suboption is equal to or greater than 3, PROC OPTMODEL issues a message for each of the nodes and links that it discards until the number of messages issued during problem generation reaches the value of the MSGLIMIT= option in the PROC OPTMODEL statement. If the value of the LOGLEVEL= suboption is greater than 0, PROC OPTMODEL also issues a summary that shows the total count of discarded nodes and links from each input array or set.

The following statements call PROC OPTMODEL and declare a five-node complete undirected graph; a subset of links that contains all links between nodes 1, 2, 3, and 4; and a subset of nodes that contains nodes 3, 4, and 5:

```
proc optmodel;
   set NODES = 1..5;
   set LINKS = {vi in NODES, vj in NODES: vi < vj};
   num distance {<vi,vj> in LINKS} = 10*vi + vj;

   set <num,num> TOUR;

   /* Build a link set using only nodes 1..4 nodes */
   set <num,num> LINKS_BETWEEN_1234 = {vi in 1..3, vj in (vi+1)..4};
   /* Build a node subset consisting of nodes 3..5 */
   set NODES_345 = 3..5;
```

After the sets are declared, the statements in the following steps solve several traveling salesman problems (TSPs) on subgraphs. For more information about TSPs, see the section "Traveling Salesman Problem" on page 452.

1. The first SOLVE statement solves a TSP on the original graph. Note that the links in the tour (see Figure 9.8) are returned with the same orientation that you provide in the input. For example, the second step on the tour goes from node 4 to node 2 using link $(2, 4)$. This guarantees that you do not need to do extra processing of output to check for link orientation. You can just use the output directly.

```
/* Implicit network 1: solve over nodes 1..5 -- The original network*/
solve with NETWORK /
   links=( weight=distance )
   out=( tour=TOUR )
   tsp
;
put TOUR=;
```

As shown in Figure 9.8, all links implied by the WEIGHT= suboption of the LINKS= option become part of the graph.

Figure 9.8 SOLVE WITH NETWORK Log: Traveling Salesman Tour of an Unfiltered Graph

```
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 10.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 111 using 0.00 (cpu:
      0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
NOTE: Optimal.
NOTE: Objective = 111.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
TOUR={<1,4>,<2,4>,<2,3>,<3,5>,<1,5>}
```

To access the objective value of a network problem, use the _OROPTMODEL_NUM_ predefined array. The network solver ignores the _OBJ_ predefined symbol, which is part of the current named problem. The current named problem is independent of the network solver, because the network solver uses sets and numeric arrays for input and output. For more information, see the sections "Multiple Subproblems" on page 150 and "Solver Status Parameters" on page 161 in Chapter 5, "The OPTMODEL Procedure."

```
put _OROPTMODEL_NUM_['OBJECTIVE'];
```

2. The next SOLVE statement solves a TSP on the subgraph that is defined by the link set LINKS_BETWEEN_1234.

```
/* Filter on LINKS: solve over nodes 1..4 */
solve with NETWORK /
   links=( weight=distance )
   subgraph=( links=LINKS_BETWEEN_1234 )
   out=( tour=TOUR )
   tsp
;
put TOUR=;
```

As shown in Figure 9.9, the network solver now ignores node 5.

Figure 9.9 SOLVE WITH NETWORK Log: Traveling Salesman Tour over Nodes $N = \{1, 2, 3, 4\}$

```
111
NOTE: The SUBGRAPH= option filtered 4 elements from 'distance.'
NOTE: The number of nodes in the input graph is 4.
NOTE: The number of links in the input graph is 6.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 74 using 0.00 (cpu:
      0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
NOTE: Optimal.
NOTE: Objective = 74.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
TOUR={<1,3>,<2,3>,<2,4>,<1,4>}
```

3. The next SOLVE statement solves a TSP on the subgraph that is defined by the node set NODES_345.

```
/* Filter on NODES: solve over nodes 3..5 */
solve with NETWORK /
   links=( weight=distance )
   subgraph=( nodes=NODES_345 )
   out=( tour=TOUR )
   tsp
;
put TOUR=;
```

As shown in Figure 9.10, the network solver now ignores nodes 1 and 2, along with any links incident to them.

Figure 9.10 SOLVE WITH NETWORK Log: Traveling Salesman Tour over Nodes $N = \{3, 4, 5\}$

```
NOTE: The SUBGRAPH= option filtered 7 elements from 'distance.'
NOTE: The number of nodes in the input graph is 3.
NOTE: The number of links in the input graph is 3.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 114 using 0.00 (cpu:
      0.00) seconds.
NOTE: Optimal.
NOTE: Objective = 114.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
TOUR={<3,4>,<4,5>,<3,5>}
```

4. The next SOLVE statement attempts to solve a TSP on the subgraph that is defined by the node set NODES_345 and the link set that is defined by the links on the nodes $\{1, 2, 3, 4\}$. This subgraph creates an infeasible instance because the links $\{(1, 5), (2, 5), (3, 5), (4, 5)\}$ that were defined in the original graph have been filtered out. Thus, node 5 is disconnected and no tour can exist.

```
/* Explicit nodes and links: semantic error over nodes 1..5
 * Links <u,5> are undefined and no documented default exists. */
solve with NETWORK /
   links=( weight=distance )
   subgraph=( nodes=NODES_345 links=LINKS_BETWEEN_1234 )
   out=( tour=TOUR )
   tsp
;
```

As shown in Figure 9.11, the network solver identifies that no tour exists over the surviving nodes and links.

Figure 9.11 SOLVE WITH NETWORK Log: Infeasible Traveling Salesman Problem after Filtering

```
NOTE: The SUBGRAPH= option filtered 4 elements from 'distance.'
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 6.
NOTE: The number of singleton nodes in the input graph is 1.
NOTE: Processing the traveling salesman problem.
NOTE: Infeasible.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
```

5. The last SOLVE statement uses the LINKS= suboption of the OUT= option to capture exactly which nodes and links were generated and with which attributes. In this case, because the only attribute defined is link weight, the set LINKS_OUT has tuples of length three.

```
/* make room for tail, head, and weight */
set<num,num,num> LINKS_OUT;
solve with NETWORK /
   links=( weight=distance )
   subgraph=( nodes=NODES_345 links=LINKS_BETWEEN_1234 )
   out=( tour=TOUR links=LINKS_OUT )
   tsp
;
put LINKS_OUT=;
quit;
```

As shown in Figure 9.12, the network solver can return the graph after filtering. This feature can sometimes help you identify why you might get counterintuitive results.

Figure 9.12 SOLVE WITH NETWORK Log: Remaining Links after Filtering

```
NOTE: The SUBGRAPH= option filtered 4 elements from 'distance.'
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 6.
NOTE: The number of singleton nodes in the input graph is 1.
NOTE: Processing the traveling salesman problem.
NOTE: Infeasible.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
LINKS_OUT={<1,2,12>,<1,3,13>,<1,4,14>,<2,3,23>,<2,4,24>,<3,4,34>}
NOTE: The PROCEDURE OPTMODEL printed pages 5-14.
```

Numeric Limitations

Extremely large or extremely small numerical values might cause computational difficulties for some of the algorithms in the network solver. For this reason, each algorithm restricts the magnitude of the data values to a particular threshold number. If the user data values exceed this threshold, the network solver issues an error message. The value of the threshold limit is different for each algorithm and depends on the operating environment. The threshold limits are listed in Table 9.20, where M is defined as the largest absolute value representable in your operating environment.

Table 9.20 Threshold Limits by Algorithm

	Graph Links				Graph Nodes	
Algorithm	weight	weight2	lower	upper	weight	weight2
CYCLE	$\sqrt{M}$				$\sqrt{M}$	
LINEAR_ASSIGNMENT	$\sqrt{M}$					
MINCOSTFLOW	1e15		1e15	1e15	1e15	1e15
MINCUT	$\sqrt{M}$					
MINSPANTREE	$\sqrt{M}$					
SHORTPATH	$\sqrt{M}$	$\sqrt{M}$				
TSP	1e20					

To obtain these limits, use the SAS CONSTANT function. For example, the following PROC OPTMODEL code assigns $\sqrt{M}$ to a variable x and prints that value to the log:

```
proc optmodel;
   num c = constant('SQRTBIG');
   put c=;
quit;
```

Missing Values

A missing value has no valid interpretation for most of the algorithms in the network solver. If the user data contain a missing value, the network solver issues an error message. There is only one exception: the minimum-cost network flow algorithm interprets a missing value in the lower or upper bound option as the default bound value. For more information about this algorithm, see the section "Minimum-Cost Network Flow" on page 424.

Negative Link Weights

For certain algorithms in the network solver, a negative link weight is not allowed. The following algorithms issue an error message if a negative link weight is provided:

- MINCUT

Biconnected Components and Articulation Points

A *biconnected component* of a graph $G = (N, A)$ is a connected subgraph that cannot be broken into disconnected pieces by deleting any single node (and its incident links). An *articulation point* is a node of a graph whose removal would cause an increase in the number of connected components. Articulation points can be important when you analyze any graph that represents a communications network. Consider an articulation point $i \in N$ which, if removed, disconnects the graph into two components C^1 and C^2. All paths in G between some nodes in C^1 and some nodes in C^2 must pass through node i. In this sense, articulation points are critical to communication. Examples of where articulation points are important are airline hubs, electric circuits, network wires, protein bonds, traffic routers, and numerous other industrial applications.

In the network solver, you can find biconnected components and articulation points of an input graph by invoking the BICONCOMP option. This algorithm works only with undirected graphs.

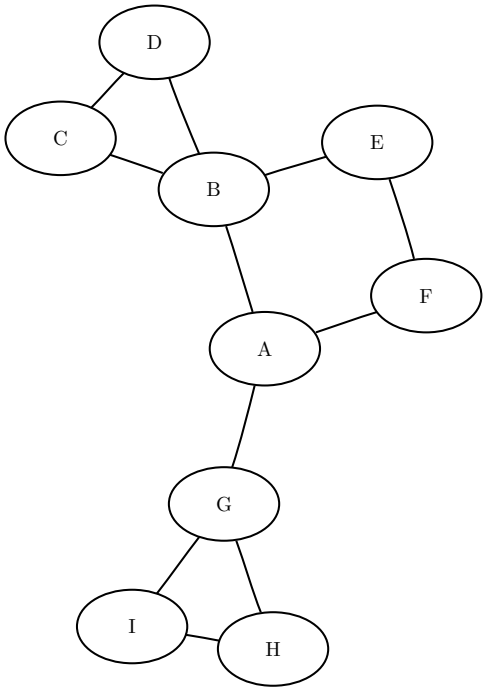
The results for the biconnected components algorithm are written to the link-indexed numeric array that is specified in the BICONCOMP= suboption of the OUT= option. For each link in the links array, the value in this array identifies its component. The component identifiers are numbered sequentially starting from 1. The articulation points are written to the set that is specified in the ARTPOINTS= suboption of the OUT= option.

The algorithm that the network solver uses to compute biconnected components is a variant of depth-first search (Tarjan 1972). This algorithm runs in time $O(|N| + |A|)$ and therefore should scale to very large graphs.

Biconnected Components of a Simple Undirected Graph

This section illustrates the use of the biconnected components algorithm on the simple undirected graph G that is shown in Figure 9.13.

Figure 9.13 A Simple Undirected Graph G



The undirected graph G can be represented by the links data set LinkSetInBiCC as follows:

```
data LinkSetInBiCC;
   input from $ to $ @@;
   datalines;
A B  A F  A G  B C  B D
B E  C D  E F  G I  G H
H I
;
```

The following statements calculate the biconnected components and articulation points and output the results in the data sets LinkSetOut and NodeSetOut:

```
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetInBiCC into LINKS=[from to];
   set NODES = union{<i,j> in LINKS} {i,j};
   num bicomponent{LINKS};
   set<str> ARTPOINTS;

   solve with NETWORK /
      loglevel  = moderate
      links     = (include=LINKS)
      biconcomp
      out       = (biconcomp=bicomponent artpoints=ARTPOINTS)
   ;

   print bicomponent;
   put ARTPOINTS;
   create data LinkSetOut from [from to] biconcomp=bicomponent;
   create data NodeSetOut from [node]=ARTPOINTS artpoint=1;
quit;
```

The data set LinkSetOut now contains the biconnected components of the input graph, as shown in Figure 9.14.

Figure 9.14 Biconnected Components of a Simple Undirected Graph

from	to	biconcomp
A	B	2
A	F	2
A	G	4
B	C	1
B	D	1
B	E	2
C	D	1
E	F	2
G	I	3
G	H	3
H	I	3

In addition, the data set NodeSetOut contains the articulation points of the input graph, as shown in Figure 9.15.

Figure 9.15 Articulation Points of a Simple Undirected Graph

node	artpoint
A	1
B	1
G	1

The biconnected components are shown graphically in Figure 9.16 and Figure 9.17.

Figure 9.16 Biconnected Components C^1 and C^2

$C^1 = \{B, C, D\}$ $C^2 = \{A, B, E, F\}$

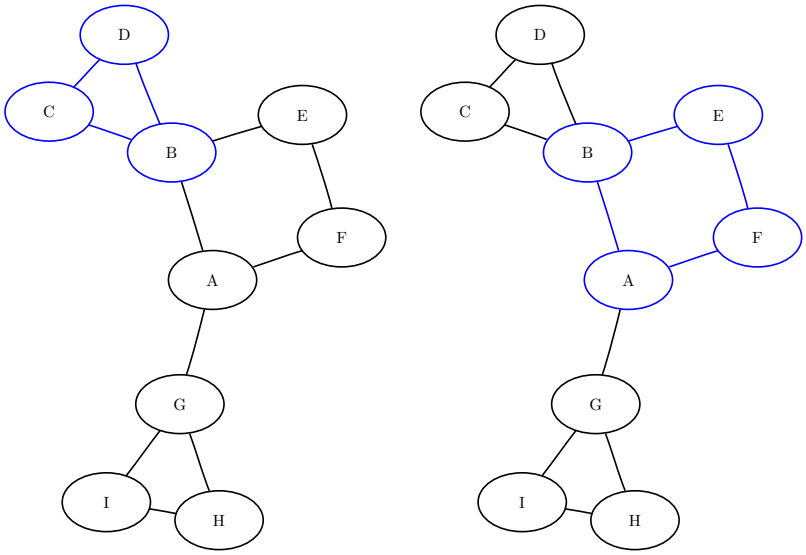
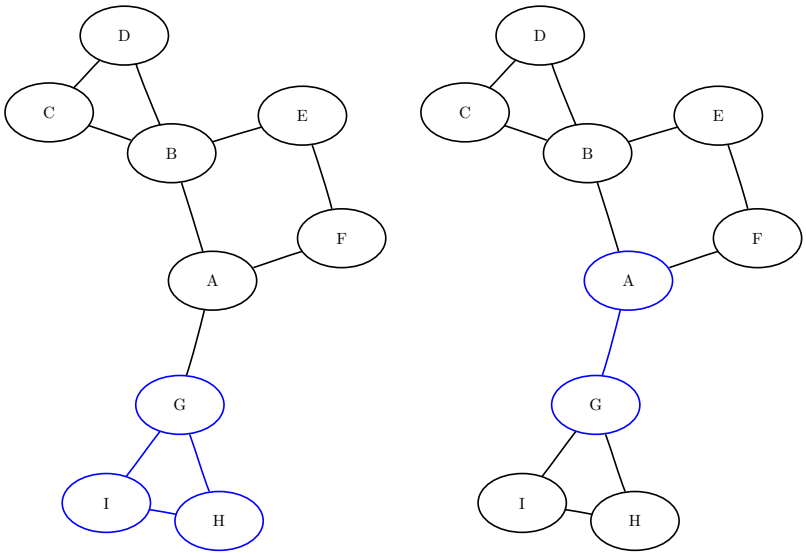


Figure 9.17 Biconnected Components C^3 and C^4

$C^3 = \{G, H, I\}$ $C^4 = \{A, G\}$



For a more detailed example, see "Example 9.1: Articulation Points in a Terrorist Network" on page 462.

Clique

A *clique* of a graph $G = (N, A)$ is an induced subgraph that is a complete graph. Every node in a clique is connected to every other node in that clique. A *maximal clique* is a clique that is not a subset of the nodes of any larger clique. That is, it is a set C of nodes such that every pair of nodes in C is connected by a link and every node not in C is missing a link to at least one node in C. The number of maximal cliques in a particular graph can be very large and can grow exponentially with every node added. Finding cliques in graphs has applications in numerous industries including bioinformatics, social networks, electrical engineering, and chemistry.

You can find the maximal cliques of an input graph by invoking the CLIQUE= option. The clique algorithm works only with undirected graphs.

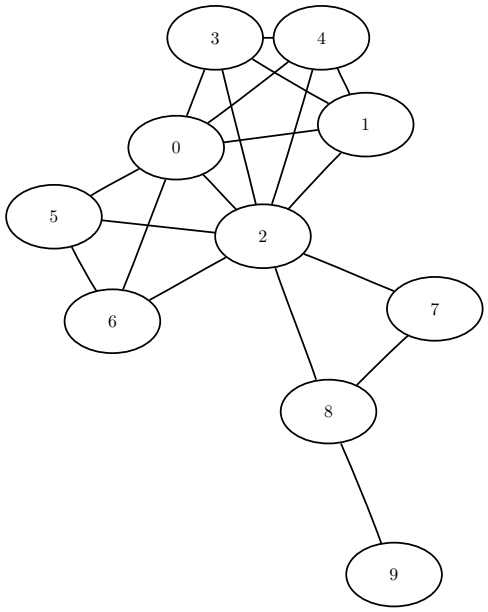
The results for the clique algorithm are written to the set that is specified in the CLIQUES= suboption of the OUT= option. Each node of each clique is listed in the set along with a clique ID (the first argument of the tuple) to identify the clique to which it belongs. A node can appear multiple times in this set if it belongs to multiple cliques.

The algorithm that the network solver uses to compute maximal cliques is a variant of the Bron-Kerbosch algorithm (Bron and Kerbosch 1973; Harley 2003). Enumerating all maximal cliques is NP-hard, so this algorithm typically does not scale to very large graphs.

Maximal Cliques of a Simple Undirected Graph

This section illustrates the use of the clique algorithm on the simple undirected graph G that is shown in Figure 9.18.

Figure 9.18 A Simple Undirected Graph G



The undirected graph G can be represented by the following links data set LinkSetIn:

```
data LinkSetIn;
   input from to @@;
   datalines;
0 1  0 2  0 3  0 4  0 5
0 6  1 2  1 3  1 4  2 3
2 4  2 5  2 6  2 7  2 8
3 4  5 6  7 8  8 9
;
```

The following statements calculate the maximal cliques, output the results in the data set Cliques, and use the CARD function and SLICE operator as a convenient way to compute the clique sizes, which are output to a data set called CliqueSizes:

```
proc optmodel;
   set<num,num> LINKS;
   read data LinkSetIn into LINKS=[from to];
   set<num,num> CLIQUES;

   solve with NETWORK /
      links  = (include=LINKS)
      clique
      out    = (cliques=CLIQUES)
   ;

   put CLIQUES;
   create data Cliques from [clique node]=CLIQUES;
   num num_cliques = card(setof {<cid,node> in CLIQUES} cid);
   set CLIQUE_IDS = 1..num_cliques;
   num size {cid in CLIQUE_IDS} = card(slice(<cid,*>, CLIQUES));
   create data CliqueSizes from [clique] size;
quit;
```

The data set Cliques now contains the maximal cliques of the input graph; it is shown in Figure 9.19.

Figure 9.19 Maximal Cliques of a Simple Undirected Graph

clique	node
1	0
1	2
1	1
1	3
1	4
2	0
2	2
2	5
2	6
3	2
3	8
3	7
4	8
4	9

In addition, the data set CliqueSizes contains the number of nodes in each clique; it is shown in Figure 9.20.

Figure 9.20 Sizes of Maximal Cliques of a Simple Undirected Graph

clique	size
1	5
2	4
3	3
4	2

The maximal cliques are shown graphically in Figure 9.21 and Figure 9.22.

Figure 9.21 Maximal Cliques C^1 and C^2

$C^1 = \{0, 1, 2, 3, 4\}$ $C^2 = \{0, 2, 5, 6\}$

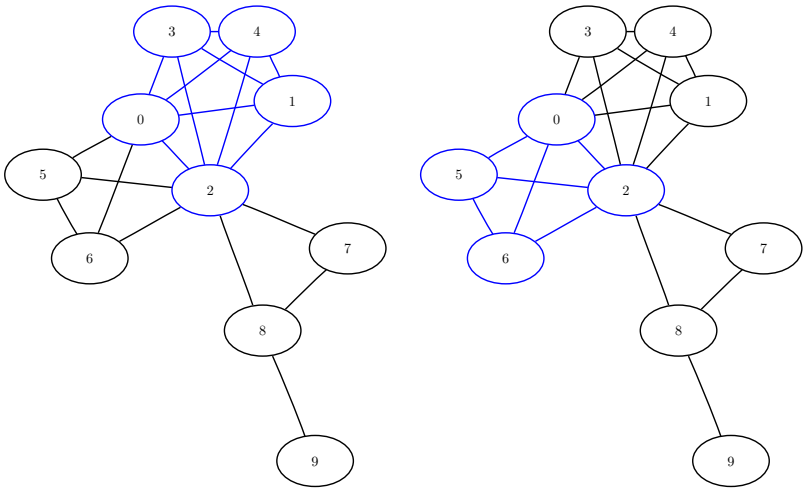
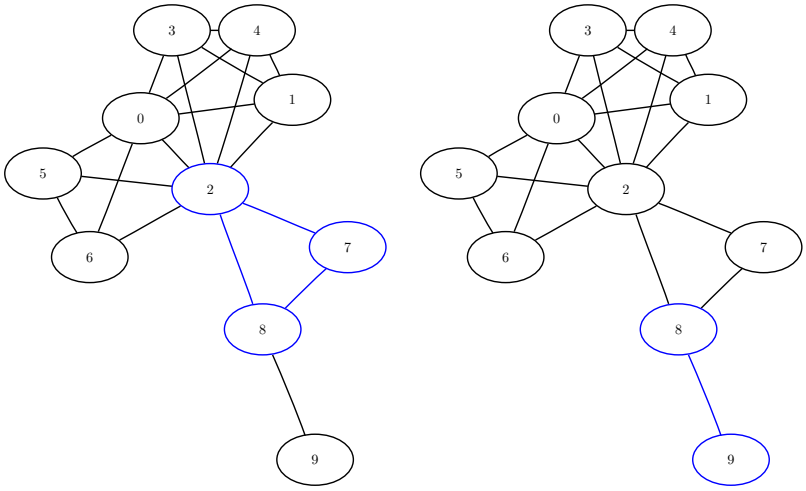


Figure 9.22 Maximal Cliques C^3 and C^4

$C^3 = \{2, 7, 8\}$ $C^4 = \{8, 9\}$



Connected Components

A *connected component* of a graph is a set of nodes that are all reachable from each other. That is, if two nodes are in the same component, then there exists a path between them. For a directed graph, there are two types of components: a *strongly connected component* has a directed path between any two nodes, and a *weakly connected component* ignores direction and requires only that a path exist between any two nodes.

In the network solver, you can invoke connected components by using the CONCOMP= option.

There are two main algorithms for finding connected components in an undirected graph: a depth-first search algorithm (ALGORITHM=DFS) and a union-find algorithm (ALGORITHM=UNION_FIND). For a graph

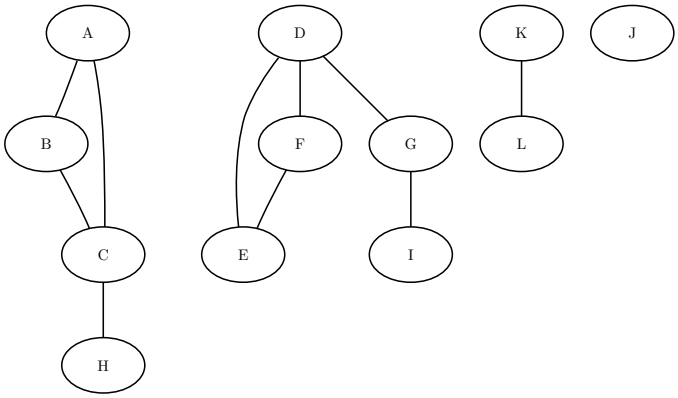
$G = (N, A)$, both algorithms run in time $O(|N| + |A|)$ and can usually scale to very large graphs. The default is the union-find algorithm. For directed graphs, only the depth-first search algorithm is available.

The results of the connected components algorithm are written to the node-indexed numeric array that you specify in the CONCOMP= suboption of the OUT= option. For each node in the set, the value of this array identifies its component. The component identifiers are numbered sequentially starting from 1.

Connected Components of a Simple Undirected Graph

This section illustrates the use of the connected components algorithm on the simple undirected graph G that is shown in Figure 9.23.

Figure 9.23 A Simple Undirected Graph G



The undirected graph G can be represented by the following links data set, LinkSetIn:

```
data LinkSetIn;
   input from $ to $ @@;
   datalines;
A B  A C  B C  C H  D E  D F  D G  F E  G I  K L
;
```

The following statements calculate the connected components and output the results in the data set NodeSetOut:

```
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetIn into LINKS=[from to];
   set NODES = union {<i,j> in LINKS} {i,j};
   num component{NODES};

   solve with NETWORK /
      links   = (include=LINKS)
      concomp
      out     = (concomp=component)
   ;

   print component;
   create data NodeSetOut from [node] concomp=component;
quit;
```

The data set NodeSetOut contains the connected components of the input graph and is shown in Figure 9.24.

Figure 9.24 Connected Components of a Simple Undirected Graph

node	concomp
A	1
B	1
C	1
H	1
D	2
E	2
F	2
G	2
I	2
K	3
L	3

Notice that the graph is defined by using only the links array. As seen in Figure 9.23, this graph also contains a singleton node labeled J, which has no associated links. By definition, this node defines its own component. But because the input graph was defined by using only the links array, node J did not show up in the results data set. To define a graph by using nodes that have no associated links, you should also define the input nodes set. In this case, you can define a nodes data set NodeSetIn as follows:

```
data NodeSetIn;
   input node $ @@;
   datalines;
A  B  C  D  E  F  G  H  I  J  K  L
;
```

You could also have defined the set directly in PROC OPTMODEL, but in this case, a separate data set nicely preserves the independence between the model and the data.

Now, when you calculate the connected components, you define the input graph by using both the nodes input data set and the links input data set:

```
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetIn into LINKS=[from to];
   set<str> NODES;
   read data NodeSetIn into NODES=[node];
   num component{NODES};

   solve with NETWORK /
      links   = (include=LINKS)
      nodes   = (include=NODES)
      concomp
      out     = (concomp=component)
   ;

   print component;
   create data NodeSetOut from [node] concomp=component;
quit;
```

The resulting data set, NodeSetOut, includes the singleton node J as its own component, as shown in Figure 9.25.

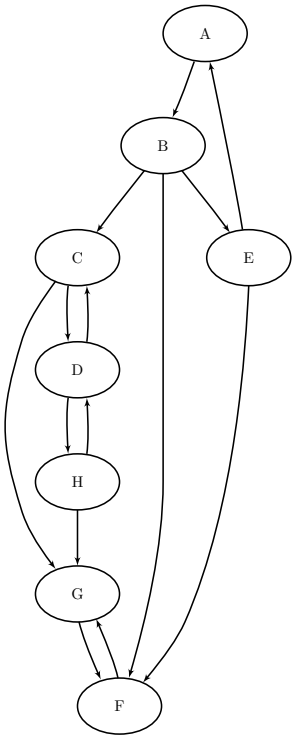
Figure 9.25 Connected Components of a Simple Undirected Graph

node	concomp
A	1
B	1
C	1
D	2
E	2
F	2
G	2
H	1
I	2
J	3
K	4
L	4

Connected Components of a Simple Directed Graph

This section illustrates the use of the connected components algorithm on the simple directed graph G that is shown in Figure 9.26.

Figure 9.26 A Simple Directed Graph G



The directed graph G can be represented by the following links data set, LinkSetIn:

```
data LinkSetIn;
   input from $ to $ @@;
   datalines;
A B  B C  B E  B F  C G
C D  D C  D H  E A  E F
F G  G F  H G  H D
;
```

The following statements calculate the connected components and output the results in the data set NodeSetOut:

```
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetIn into LINKS=[from to];
   set NODES = union {<i,j> in LINKS} {i,j};
   num component{NODES};

   solve with NETWORK /
      graph_direction = directed
      links           = (include=LINKS)
      concomp
      out             = (concomp=component)
   ;

   print component;
   create data NodeSetOut from [node] concomp=component;
quit;
```

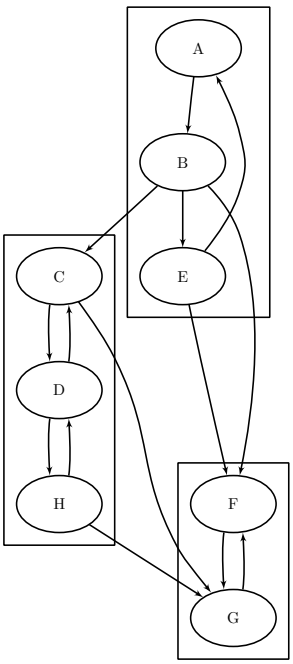
The data set NodeSetOut, shown in Figure 9.27, now contains the connected components of the input graph.

Figure 9.27 Connected Components of a Simple Directed Graph

node	concomp
A	3
B	3
C	2
E	3
F	1
G	1
D	2
H	2

The connected components are represented graphically in Figure 9.28.

Figure 9.28 Strongly Connected Components of Graph G



Cycle

A *path* in a graph is a sequence of nodes, each of which has a link to the next node in the sequence. An *elementary cycle* is a path in which the start node and end node are the same and otherwise no node appears more than once in the sequence.

In the network solver, you can find you can find (or just count) the elementary cycles of an input graph by invoking the CYCLE= algorithm option. To find the cycles and report them in a set, use the CYCLES= suboption in the OUT= option. You do not need to use the CYCLES= suboption to simply count the cycles.

For undirected graphs, each link represents two directed links. For this reason, the following cycles are filtered out: trivial cycles ($A \rightarrow B \rightarrow A$) and duplicate cycles that are found by traversing a cycle in both directions ($A \rightarrow B \rightarrow C \rightarrow A$ and $A \rightarrow C \rightarrow B \rightarrow A$).

The results of the cycle detection algorithm are written to the set that you specify in the CYCLES= suboption in the OUT= option. Each node of each cycle is listed in the CYCLES= set along with a cycle ID (the first argument of the tuple) to identify the cycle to which it belongs. The second argument of the tuple defines the order (sequence) of the node in the cycle.

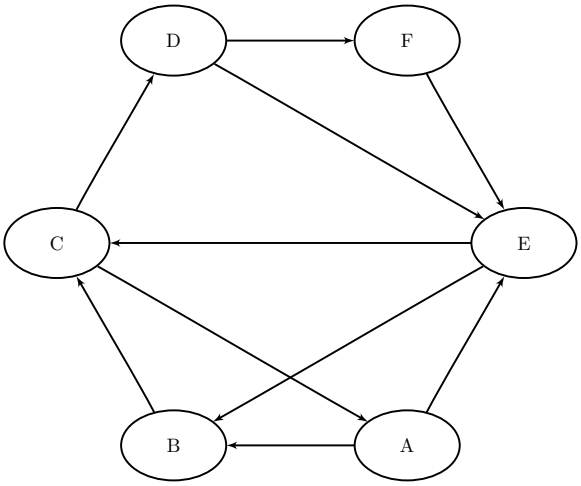
The algorithm that the network solver uses to compute all cycles is a variant of the algorithm in Johnson (1975). This algorithm runs in time $O((|N| + |A|)(c + 1))$, where c is the number of elementary cycles in the graph. So the algorithm should scale to large graphs that contain few cycles. However, some graphs can have a very large number of cycles, so the algorithm might not scale.

If MODE=ALL_CYCLES and there are many cycles, the CYCLES= set can become very large. It might be beneficial to check the number of cycles before you try to create the CYCLES= set. When you specify MODE=FIRST_CYCLE, the algorithm returns the first cycle that it finds and stops processing. This should run relatively quickly. For large-scale graphs, the MINLINKWEIGHT= and MAXLINKWEIGHT= suboptions might increase the computation time.

Cycle Detection of a Simple Directed Graph

This section provides a simple example of using the cycle detection algorithm on the simple directed graph *G* that is shown in Figure 9.29. Two other examples are "Example 9.2: Cycle Detection for Kidney Donor Exchange" on page 464, which shows the use of cycle detection for optimizing a kidney donor exchange, and "Example 9.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System" on page 476, which shows an application of cycle detection to dependencies between bug reports.

Figure 9.29 A Simple Directed Graph *G*



The directed graph *G* can be represented by the following links data set, LinkSetIn:

```
data LinkSetIn;
   input from $ to $ @@;
   datalines;
A B  A E  B C  C A  C D
D E  D F  E B  E C  F E
;
```

The following statements check whether the graph has a cycle:

```
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetIn into LINKS=[from to];
   set<num,num,str> CYCLES;

   solve with NETWORK /
      graph_direction = directed
      links           = (include=LINKS)
      cycle           = (mode=first_cycle)
   ;
quit;
```

The result is written to the log of the procedure, as shown in Figure 9.30.

Figure 9.30 Network Solver Log: Check the Existence of a Cycle in a Simple Directed Graph

```
NOTE: There were 10 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 10.
NOTE: Processing cycle detection.
NOTE: The graph does have a cycle.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
```

The following statements count the number of cycles in the graph:

```
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetIn into LINKS=[from to];
   set<num,num,str> CYCLES;

   solve with NETWORK /
      graph_direction = directed
      links           = (include=LINKS)
      cycle           = (mode=all_cycles)
   ;
quit;
```

The result is written to the log of the procedure, as shown in Figure 9.31.

Figure 9.31 Network Solver Log: Count the Number of Cycles in a Simple Directed Graph

```
NOTE: There were 10 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 10.
NOTE: Processing cycle detection.
NOTE: The graph has 7 cycles.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
```

The following statements return the first cycle found in the graph:

```
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetIn into LINKS=[from to];
   set<num,num,str> CYCLES;

   solve with NETWORK /
      graph_direction = directed
      links           = (include=LINKS)
      cycle           = (mode=first_cycle)
      out             = (cycles=CYCLES)
   ;

   put CYCLES;
   create data Cycles from [cycle order node]=CYCLES;
quit;
```

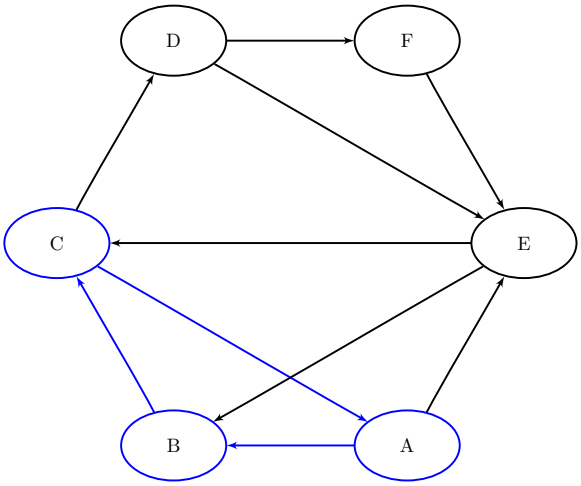
The data set Cycles now contains the first cycle found in the input graph; it is shown in Figure 9.32.

Figure 9.32 First Cycle Found in a Simple Directed Graph

cycle	order	node
1	1	A
1	2	B
1	3	C
1	4	A

The first cycle that is found in the input graph is shown graphically in Figure 9.33.

Figure 9.33 $A \rightarrow B \rightarrow C \rightarrow A$



The following statements return all the cycles in the graph:

```
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetIn into LINKS=[from to];
   set<num,num,str> CYCLES;

   solve with NETWORK /
      graph_direction = directed
      links           = (include=LINKS)
      cycle           = (mode=all_cycles)
      out             = (cycles=CYCLES)
   ;

   put CYCLES;
   create data Cycles from [cycle order node]=CYCLES;
quit;
```

The data set Cycles now contains all the cycles in the input graph; it is shown in Figure 9.34.

Figure 9.34 All Cycles in a Simple Directed Graph

cycle	order	node
1	1	A
1	2	B
1	3	C
1	4	A
2	1	A
2	2	E
2	3	B
2	4	C
2	5	A

cycle	order	node
3	1	A
3	2	E
3	3	C
3	4	A
4	1	B
4	2	C
4	3	D
4	4	E
4	5	B

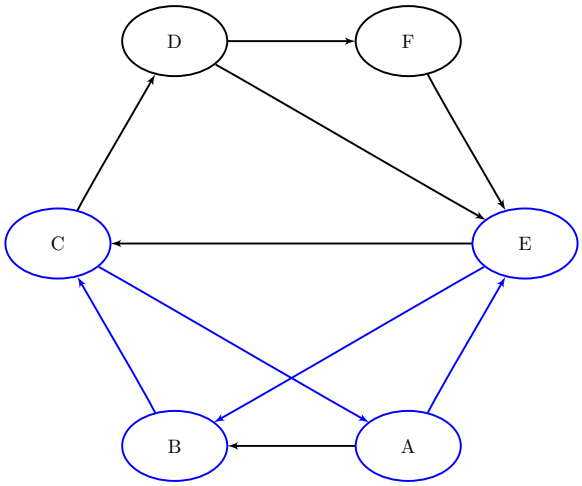
cycle	order	node
5	1	B
5	2	C
5	3	D
5	4	F
5	5	E
5	6	B
6	1	E
6	2	C
6	3	D

cycle	order	node
6	4	E
7	1	E
7	2	C
7	3	D
7	4	F
7	5	E

The six additional cycles are shown graphically in Figure 9.35 through Figure 9.37.

Figure 9.35 Cycles

$A \to E \to B \to C \to A$



$A \to E \to C \to A$

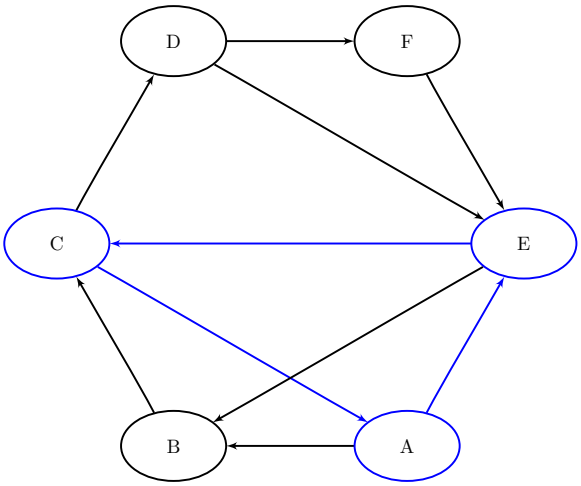


Figure 9.36 Cycles

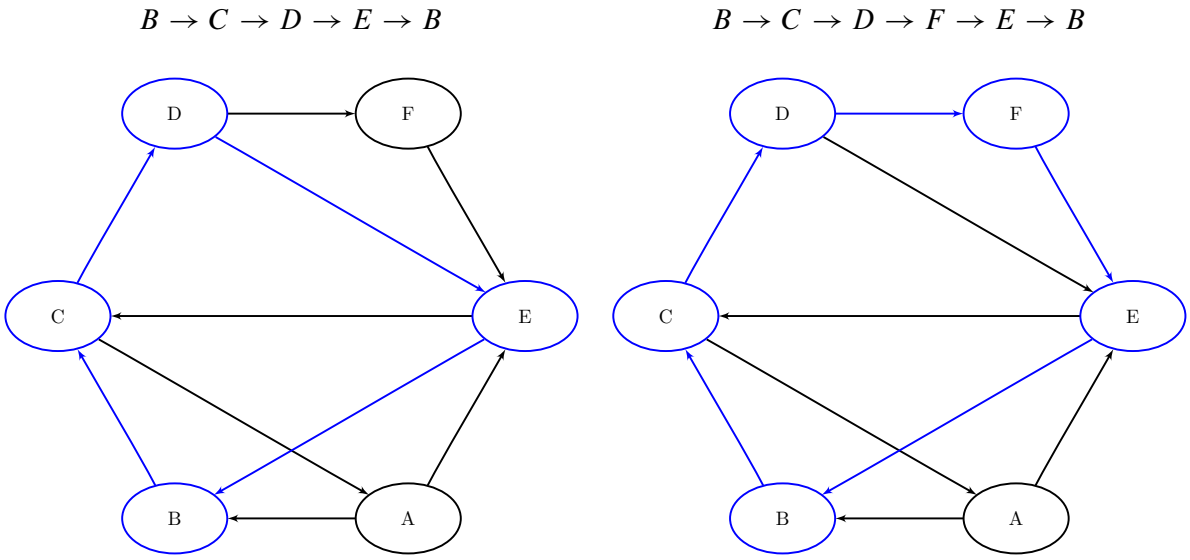
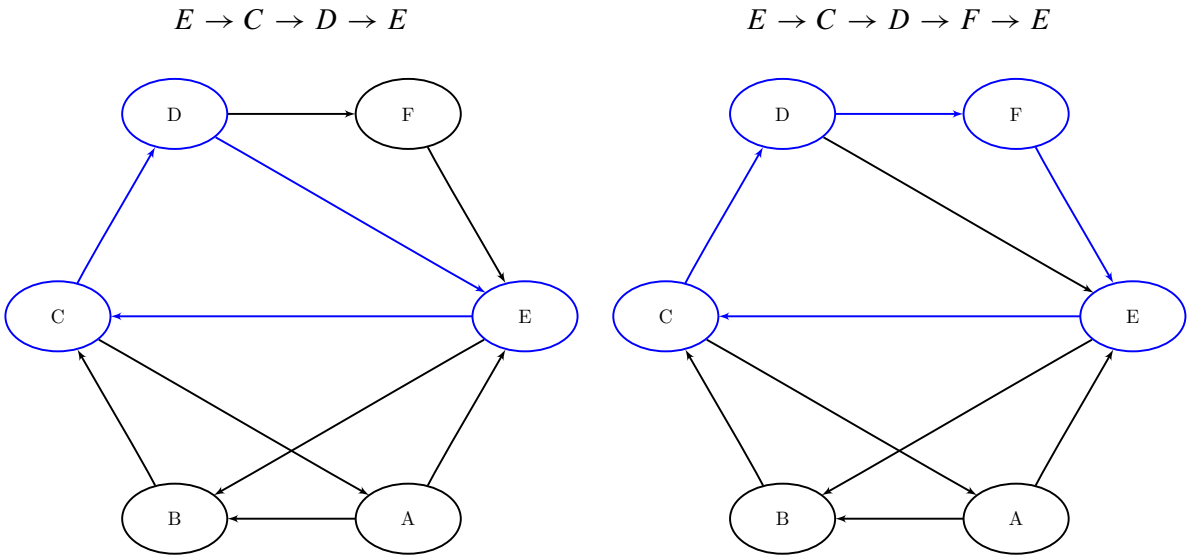


Figure 9.37 Cycles



Linear Assignment (Matching)

The *linear assignment problem* (LAP) is a fundamental problem in combinatorial optimization that involves assigning workers to tasks at minimal costs. In graph theoretic terms, the LAP is equivalent to finding a minimum-weight matching in a weighted bipartite directed graph. In a *bipartite graph*, the nodes can be divided into two disjoint sets S (workers) and T (tasks) such that every link connects a node in S to a node in T. That is, the node sets S and T are independent. The concept of assigning workers to tasks can be generalized to the assignment of any abstract object from one group to some abstract object from a second group.

The linear assignment problem can be formulated as an integer programming optimization problem. The form of the problem depends on the sizes of the two input sets, S and T. Let A represent the set of possible assignments between sets S and T. In the bipartite graph, these assignments are the links. If $|S| \geq |T|$, then the following optimization problem is solved:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in A} c_{ij}x_{ij} \\
\text{subject to} \quad & \sum_{(i,j)\in A} x_{ij} \leq 1 \quad i \in S \\
& \sum_{(i,j)\in A} x_{ij} = 1 \quad j \in T \\
& x_{ij} \in \{0,1\} \qquad (i,j) \in A
\end{aligned}$$

This model allows for some elements of set S (workers) to go unassigned (if $|S| > |T|$).

If $|S| < |T|$, then the following optimization problem is solved:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in A} c_{ij}x_{ij} \\
\text{subject to} \quad & \sum_{(i,j)\in A} x_{ij} = 1 \quad i \in S \\
& \sum_{(i,j)\in A} x_{ij} \leq 1 \quad j \in T \\
& x_{ij} \in \{0,1\} \qquad (i,j) \in A
\end{aligned}$$

This model allows for some elements of set T (tasks) to go unassigned.

In the network solver, you can invoke the linear assignment problem solver by using the LINEAR_ASSIGNMENT option. The algorithm that the network solver uses for solving a LAP is based on augmentation of shortest paths (Jonker and Volgenant 1987). This algorithm can be applied as long as the graph is bipartite.

The resulting assignment (or matching) is contained in the set that is specified in the ASSIGNMENTS= suboption of the OUT= option.

For a detailed example, see "Example 9.3: Linear Assignment Problem for Minimizing Swim Times" on page 468.

Minimum-Cost Network Flow

The *minimum-cost network flow problem* (MCF) is a fundamental problem in network analysis that involves sending flow over a network at minimal cost. Let $G = (N, A)$ be a directed graph. For each link $(i, j) \in A$, associate a cost per unit of flow, designated by c_{ij}. The demand (or supply) at each node $i \in N$ is designated as b_i, where $b_i \geq 0$ denotes a supply node and $b_i < 0$ denotes a demand node. These values must be within $[b_i^l, b_i^u]$. Define decision variables x_{ij} that denote the amount of flow sent from node i to node j. The amount of flow that can be sent across each link is bounded to be within $[l_{ij}, u_{ij}]$. The problem can be modeled as a

linear programming problem as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in A} c_{ij}x_{ij} \\
\text{subject to} \quad & b_i^l \le \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} \le b_i^u \quad i \in N \\
& l_{ij} \le x_{ij} \le u_{ij} \qquad\qquad (i,j) \in A
\end{aligned}$$

When $b_i = b_i^l = b_i^u$ for all nodes $i \in N$, the problem is called a *pure network flow problem*. For these problems, the sum of the supplies and demands must be equal to 0 to ensure that a feasible solution exists.

In the network solver, you can invoke the minimum-cost network flow solver by using the MINCOSTFLOW option.

The algorithm that the network solver uses for solving MCF is a variant of the primal network simplex algorithm (Ahuja, Magnanti, and Orlin 1993). Sometimes the directed graph G is disconnected. In this case, the problem is first decomposed into its weakly connected components, and then each minimum-cost flow problem is solved separately.

The input for the network is the standard graph input, which is described in the section "Input Data for the Network Solver" on page 398. The MCF option uses the following suboptions of the LINKS= input option that specify link-indexed numeric arrays:

- The WEIGHT= suboption defines the link cost c_{ij} per unit of flow. (The default is 0, but if the WEIGHT= suboption is not specified, then the default is 1.)
- The LOWER= suboption defines the link flow lower bound l_{ij}. (The default is 0.)
- The UPPER= suboption defines the link flow upper bound u_{ij}. (The default is ∞.)

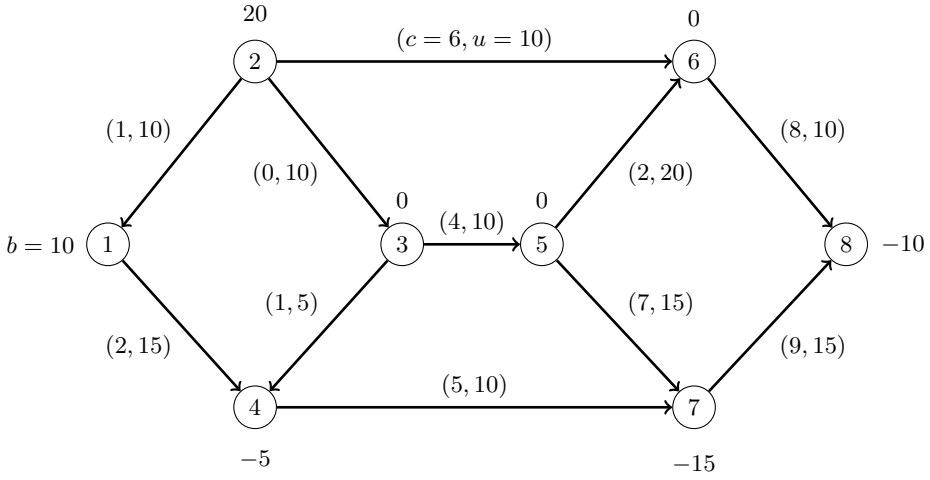
The MCF option uses the WEIGHT= suboption of the NODES= option to specify supply. The parameter is a numeric array that is positive for supply nodes and negative for demand nodes.

The resulting optimal flow through the network is written to the link-indexed numeric array that is specified in the FLOW= suboption of the OUT= option in the SOLVE WITH NETWORK statement.

Minimum Cost Network Flow for a Simple Directed Graph

The following example demonstrates how to use the network simplex algorithm to find a minimum-cost flow in a directed graph. Consider the directed graph in Figure 9.38, which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 9.38 Minimum-Cost Network Flow Problem: Data



The directed graph G can be represented by the following links data set LinkSetIn and nodes data set NodeSetIn:

```
data LinkSetIn;
   input from to weight upper;
   datalines;
1 4 2 15
2 1 1 10
2 3 0 10
2 6 6 10
3 4 1  5
3 5 4 10
4 7 5 10
5 6 2 20
5 7 7 15
6 8 8 10
7 8 9 15
;

data NodeSetIn;
   input node weight;
   datalines;
1  10
2  20
4  -5
7 -15
8 -10
;
```

You can use the following call to the network solver to find a minimum-cost flow:

```
proc optmodel;
   set <num,num> LINKS;
   num cost{LINKS};
   num upper{LINKS};
   read data LinkSetIn into LINKS=[from to] cost=weight upper;
   set NODES = union {<i,j> in LINKS} {i,j};
   num supply{NODES} init 0;
   read data NodeSetIn into [node] supply=weight;
   num flow{LINKS};

   solve with network /
      loglevel        = moderate
      logfreq         = 1
      graph_direction = directed
      links           = (upper=upper weight=cost)
      nodes           = (weight=supply)
      mcf
      out             = (flow=flow)
   ;

   print flow;
   create data LinkSetOut from [from to] upper cost flow;
quit;
```

The progress of the procedure is shown in Figure 9.39.

Figure 9.39 Network Solver Log for Minimum-Cost Network Flow

```
NOTE: There were 11 observations read from the data set WORK.LINKSETIN.
NOTE: There were 5 observations read from the data set WORK.NODESETIN.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the minimum-cost network flow problem.
NOTE: The network has 1 connected component.
                        Primal         Primal           Dual
      Iteration      Objective  Infeasibility  Infeasibility     Time
              1   0.000000E+00   2.000000E+01   8.900000E+01     0.00
              2   0.000000E+00   2.000000E+01   8.900000E+01     0.00
              3   5.000000E+00   1.500000E+01   8.400000E+01     0.00
              4   5.000000E+00   1.500000E+01   8.300000E+01     0.00
              5   7.500000E+01   1.500000E+01   8.300000E+01     0.00
              6   7.500000E+01   1.500000E+01   7.900000E+01     0.00
              7   1.300000E+02   1.000000E+01   7.600000E+01     0.00
              8   2.700000E+02   0.000000E+00   0.000000E+00     0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: Objective = 270.
NOTE: Processing the minimum-cost network flow problem used 0.00 (cpu: 0.00)
      seconds.
NOTE: The data set WORK.LINKSETOUT has 11 observations and 5 variables.
```

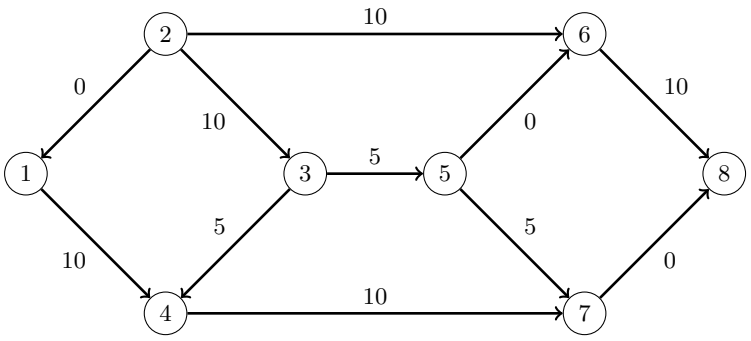
The optimal solution is displayed in Figure 9.40.

Figure 9.40 Minimum-Cost Network Flow Problem: Optimal Solution

Obs	from	to	upper	cost	flow
1	1	4	15	2	10
2	2	1	10	1	0
3	2	3	10	0	10
4	2	6	10	6	10
5	3	4	5	1	5
6	3	5	10	4	5
7	4	7	10	5	10
8	5	6	20	2	0
9	5	7	15	7	5
10	6	8	10	8	10
11	7	8	15	9	0

The optimal solution is represented graphically in Figure 9.41.

Figure 9.41 Minimum-Cost Network Flow Problem: Optimal Solution



Minimum-Cost Network Flow with Flexible Supply and Demand

Using the same directed graph shown in Figure 9.38, this example demonstrates a network that has a flexible supply and demand. Consider the following adjustments to the node bounds:

- Node 1 has an infinite supply, but it still requires at least 10 units to be sent.
- Node 4 is a throughput node that can now handle an infinite amount of demand.
- Node 8 has a flexible demand. It requires between 6 and 10 units.

You use the special missing values (.I) to represent infinity and (.M) to represent minus infinity. The adjusted node bounds can be represented by the following nodes data set:

```
data NodeSetIn;
   input node weight weight2;
   datalines;
1  10  .I
2  20  20
4  .M  -5
7 -15 -15
8 -10  -6
;
```

You can use the following call to PROC OPTMODEL to find a minimum-cost flow:

```
proc optmodel;
   set <num,num> LINKS;
   num cost{LINKS};
   num upper{LINKS};
      read data LinkSetIn into LINKS=[from to] cost=weight upper;
   set <num> NODES;
   num supply{NODES};
   num supplyUB{NODES}; /* also demand lower bound, if negative */
      read data NodeSetIn into NODES=[node] supply=weight supplyUB=weight2;
   num flow{LINKS};

   solve with NETWORK /
      direction = directed
      links     = ( upper = upper weight = cost )
      nodes     = ( weight = supply weight2 = supplyUB )
      mcf
      out       = ( flow = flow )
   ;
   print flow;
   create data LinkSetOut from [from to] upper cost flow;
quit;
```

The progress of the procedure is shown in Figure 9.42.

Figure 9.42 PROC OPTMODEL Log for Minimum-Cost Network Flow

```
NOTE: There were 11 observations read from the data set WORK.LINKSETIN.
NOTE: There were 5 observations read from the data set WORK.NODESETIN.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the minimum-cost network flow problem.
NOTE: Objective = 226.
NOTE: Processing the minimum-cost network flow problem used 0.00 (cpu: 0.00)
      seconds.
NOTE: The data set WORK.LINKSETOUT has 11 observations and 5 variables.
```

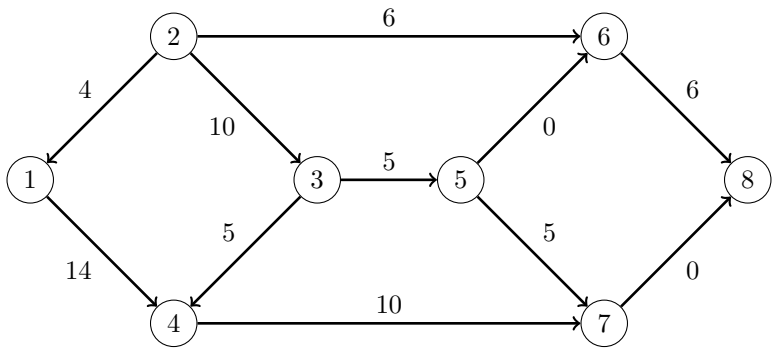
The optimal solution is displayed in Figure 9.43.

Figure 9.43 Minimum-Cost Network Flow Problem: Optimal Solution

Obs	from	to	upper	cost	flow
1	1	4	15	2	14
2	2	1	10	1	4
3	2	3	10	0	10
4	2	6	10	6	6
5	3	4	5	1	5
6	3	5	10	4	5
7	4	7	10	5	10
8	5	6	20	2	0
9	5	7	15	7	5
10	6	8	10	8	6
11	7	8	15	9	0

The optimal solution is represented graphically in Figure 9.44.

Figure 9.44 Minimum-Cost Network Flow Problem: Optimal Solution



Minimum Cut

A *cut* is a partition of the nodes of a graph into two disjoint subsets. The *cut-set* is the set of links whose *from* and *to* nodes are in different subsets of the partition. A *minimum cut* of an undirected graph is a cut whose cut-set has the smallest link metric, which is measured as follows: For an unweighted graph, the link metric is the number of links in the cut-set. For a weighted graph, the link metric is the sum of the link weights in the cut-set.

In the network solver, you can invoke the minimum cut algorithm by using the MINCUT= option. This algorithm can be used only on undirected graphs.

If the value of the MAXNUMCUTS= suboption is greater than 1, then the algorithm can return more than one set of cuts. The resulting cuts can be described in terms of partitions of the nodes of the graph or the links in the cut-sets. The example in the next section illustrates several ways to manipulate the output from the minimum cut algorithm. The node partition is specified in the PARTITIONS= suboption of the OUT= option in the SOLVE WITH NETWORK statement. Each tuple in this set has a cut ID and a node. PROC OPTMODEL provides only the smaller of the two subsets that form each partition. You can use the DIFF set

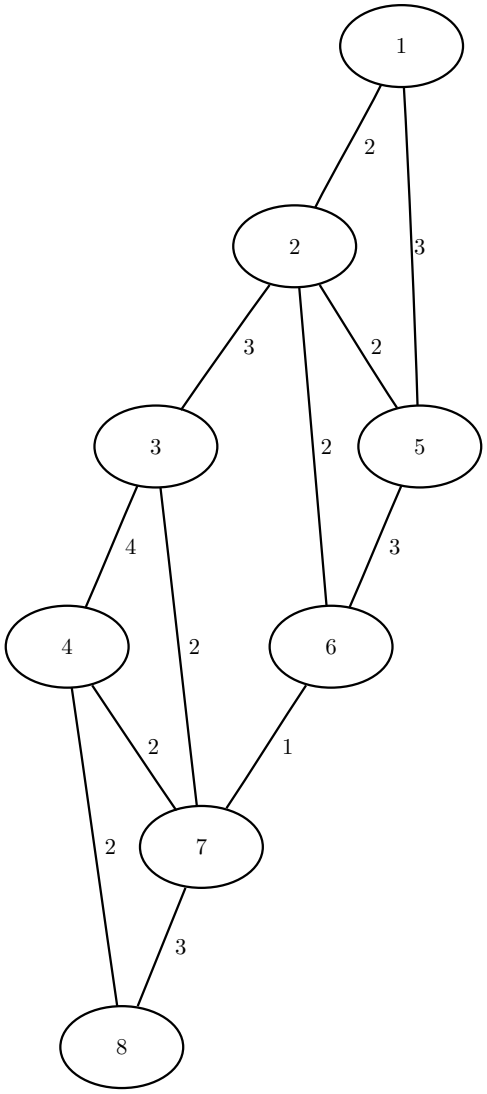
operator to get the complement of each partition. The cut-set is specified in the CUTSETS= suboption of the OUT= option. This set contains the cut ID and the corresponding list of links.

The network solver uses the Stoer-Wagner algorithm (Stoer and Wagner 1997) to compute the minimum cuts. This algorithm runs in time $O(|N||A| + |N|^2 \log |N|)$.

Minimum Cut for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 9.45.

Figure 9.45 A Simple Undirected Graph



The links data set can be represented as follows:

```
data LinkSetIn;
   input from to weight @@;
   datalines;
1 2 2  1 5 3  2 3 3  2 5 2  2 6 2
3 4 4  3 7 2  4 7 2  4 8 2  5 6 3
6 7 1  7 8 3
;
```

The following statements calculate minimum cuts in the graph and output the results in the data set MinCut:

```
proc optmodel;
   set<num,num> LINKS;
   num weight{LINKS};
   read data LinkSetIn into LINKS=[from to] weight;
   set<num> NODES = union {<i,j> in LINKS} {i,j};
   set<num,num> PARTITIONS;
   set<num,num,num> CUTSETS;

   solve with NETWORK /
      loglevel = moderate
      links    = (weight=weight)
      mincut   = (maxnumcuts=3)
      out      = (partitions=PARTITIONS cutsets=CUTSETS)
   ;
   set CUTS = setof {<cut,i,j> in CUTSETS} cut;
   num minCutWeight {cut in CUTS} = sum {<(cut),i,j> in CUTSETS} weight[i,j];
   print minCutWeight;

   for { cut in CUTS }
      put "Cut ID: "    cut
          "Partition: " ( slice( <cut,*>, PARTITIONS ) )
          "and "        ( NODES diff slice( <cut,*>, PARTITIONS ) )
          "Cut: "       ( slice( <cut,*,*>, CUTSETS ) )
          "Weight: "    minCutWeight[cut];

   create data MinCut from [mincut from to]=CUTSETS weight[from,to];
   num mincut {cut in CUTS, node in NODES} =
      if <cut,node> in PARTITIONS then 0 else 1;
   print mincut;
   create data NodeSetOut from [node]=NODES
      {cut in CUTS} <col('mincut_'||cut)=mincut[cut,node]>;
quit;
```

The progress of the procedure is shown in Figure 9.46.

Figure 9.46 Network Solver Log for Minimum Cut

```
NOTE: There were 12 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 12.
NOTE: Processing the minimum-cut problem.
NOTE: The minimum-cut algorithm found 3 cuts.
NOTE: The cut 1 has weight 4.
NOTE: The cut 2 has weight 5.
NOTE: The cut 3 has weight 5.
NOTE: Processing the minimum-cut problem used 0.00 (cpu: 0.00) seconds.
Cut ID: 1 Partition: {3,4,7,8} and {1,2,5,6} Cut: {<2,3>,<6,7>} Weight: 4
Cut ID: 2 Partition: {8} and {1,2,5,3,6,4,7} Cut: {<4,8>,<7,8>} Weight: 5
Cut ID: 3 Partition: {1} and {2,5,3,6,4,7,8} Cut: {<1,2>,<1,5>} Weight: 5
NOTE: The data set WORK.MINCUT has 6 observations and 4 variables.
NOTE: The data set WORK.NODESETOUT has 8 observations and 4 variables.
```

The data set NodeSetOut now contains the partition of the nodes for each cut, shown in Figure 9.47.

Figure 9.47 Minimum Cut Node Partition

node	mincut_1	mincut_2	mincut_3
1	1	1	0
2	1	1	1
5	1	1	1
3	0	1	1
6	1	1	1
4	0	1	1
7	0	1	1
8	0	0	1

The data set MinCut contains the links in the cut-sets for each cut. This data set is shown in Figure 9.48, which also shows each cut separately.

Figure 9.48 Minimum Cut-sets

mincut	from	to	weight
1	2	3	3
1	6	7	1
2	4	8	2
2	7	8	3
3	1	2	2
3	1	5	3

mincut=1

from	to	weight
2	3	3
6	7	1
mincut		**4**

Figure 9.48 *continued*

mincut=2

from	to	weight
4	8	2
7	8	3
mincut		**5**

mincut=3

from	to	weight
1	2	2
1	5	3
mincut		**5**
		14

Minimum Spanning Tree

A *spanning tree* of a connected undirected graph is a subgraph that is a tree that connects all the nodes together. When weights have been assigned to the links, a *minimum spanning tree* (MST) is a spanning tree whose sum of link weights is less than or equal to the sum of link weights of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a *minimum spanning forest*, which is a union of minimum spanning trees of its connected components.

In the network solver, you can invoke the minimum spanning tree algorithm by using the MINSPANTREE option. This algorithm can be used only on undirected graphs.

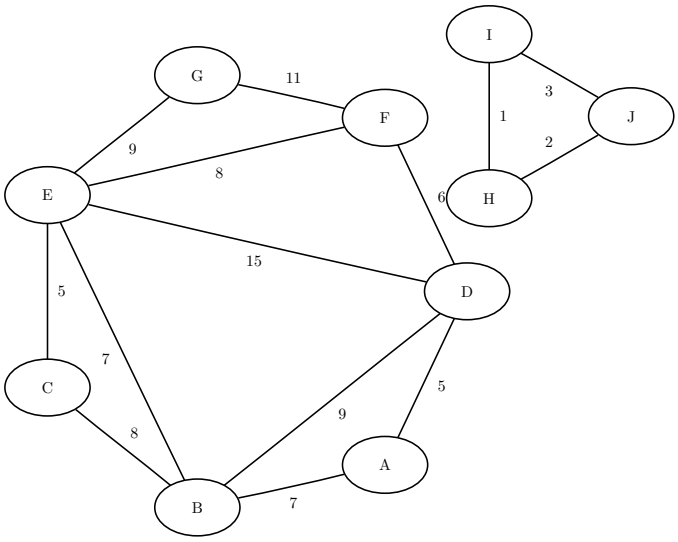
The resulting minimum spanning tree is contained in the set that is specified in the FOREST= suboption of the OUT= option in the SOLVE WITH NETWORK statement.

The network solver uses Kruskal's algorithm (Kruskal 1956) to compute the minimum spanning tree. This algorithm runs in time $O(|A| \log |N|)$ and therefore should scale to very large graphs.

Minimum Spanning Tree for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 9.49.

Figure 9.49 A Simple Undirected Graph



The links data set can be represented as follows:

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B  7  A D  5  B C 8  B D 9  B E 7
C E  5  D E 15  D F 6  E F 8  E G 9
F G 11  H I  1  I J 3  H J 2
;
```

The following statements calculate a minimum spanning forest and output the results in the data set MinSpanForest:

```
proc optmodel;
   set<str,str> LINKS;
   num weight{LINKS};
   read data LinkSetIn into LINKS=[from to] weight;
   set<str,str> FOREST;

   solve with NETWORK /
      links       = (weight=weight)
      minspantree
      out         = (forest=FOREST)
   ;

   put FOREST;
   create data MinSpanForest from [from to]=FOREST weight;
quit;
```

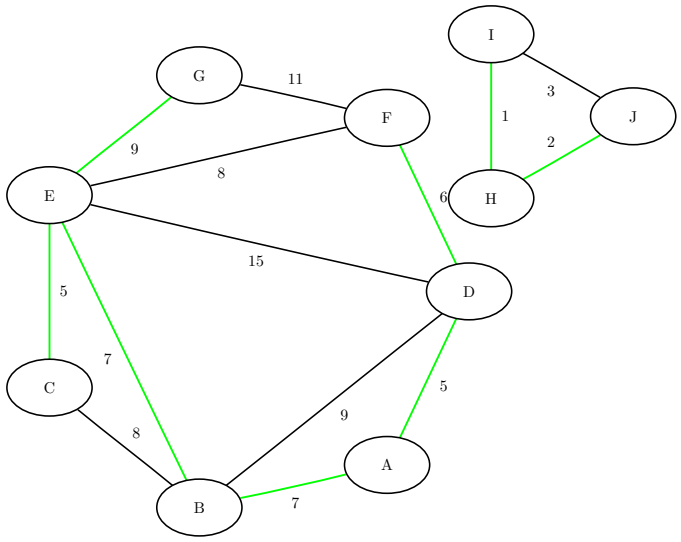
The data set MinSpanForest now contains the links that belong to a minimum spanning forest, which is shown in Figure 9.50.

Figure 9.50 Minimum Spanning Forest

from	to	weight
H	I	1
H	J	2
A	D	5
C	E	5
D	F	6
B	E	7
A	B	7
E	G	9
		42

The minimal cost links are shown in green in Figure 9.51.

Figure 9.51 Minimum Spanning Forest



For a more detailed example, see "Example 9.5: Minimum Spanning Tree for Computer Network Topology" on page 474.

Shortest Path

A *shortest path* between two nodes *u* and *v* in a graph is a path that starts at *u* and ends at *v* and has the lowest total link weight. The starting node is called the *source node*, and the ending node is called the *sink node*.

In the network solver, you can calculate shortest paths by using the SHORTPATH= option.

By default, the network solver finds shortest paths for all pairs. That is, it finds a shortest path for each possible combination of source and sink nodes. Alternatively, you can use the SOURCE= suboption to fix a particular source node and find shortest paths from the fixed source node to all possible sink nodes. Conversely, by using the SINK= suboption, you can fix a sink node and find shortest paths from all possible source nodes to the fixed sink node. By using both suboptions together, you can request one particular

shortest path for a specific source-sink pair. In addition, you can use the SOURCE= and SINK= suboptions to define a list of source-sink pairs to process. The following sections show examples of these suboptions.

Which algorithm the network solver uses to find shortest paths depends on the data. The algorithm and run-time complexity for each graph type is shown in Table 9.21.

Table 9.21 Algorithms for Shortest Paths

Graph Type	Algorithm	Complexity (per Source Node)
Unweighted	Breadth-first search	$O(\|N\| + \|A\|)$
Weighted (nonnegative)	Dijkstra's algorithm	$O(\|N\| \log \|N\| + \|A\|)$
Weighted (positive and negative allowed)	Bellman-Ford algorithm	$O(\|N\|\|A\|)$

Details for each algorithm can be found in Ahuja, Magnanti, and Orlin (1993).

For weighted graphs, the algorithm uses the parameter that is specified in the WEIGHT= suboption of the SHORTPATH= option to evaluate a path's total weight (cost).

Outputs

The shortest path algorithm produces up to two outputs. The output set that you specify in the SPPATHS= suboption contains the links of a shortest path for each source-sink pair combination. The output parameter that you specify in the SPWEIGHTS= suboption contains the total weight for the shortest path for each source-sink pair combination.

SPPATHS= Set

The SPPATHS= set contains the links present in the shortest path for each source-sink pair.

The individual links in this set always appear in the order that you provide. If you provide link (u, v) and solve a shortest path problem on an undirected graph, and that path visits node v before node u, then this set will contain link (u, v), not link (v, u), which is not part of the network. This approach simplifies indexing throughout your model. In certain use cases, especially for producing output, you might prefer to see the links in the order in which the nodes are visited. For one way to do that, see "Example 9.7: Traveling Salesman Tour through US Capital Cities" on page 479.

For large graphs and a large requested number of source-sink pairs, this set can be extremely large. For extremely large graphs, generating the output can sometimes take longer than computing the shortest paths. For example, using the US road network data for the state of New York, the data contain a directed graph that has 264,346 nodes. Finding the shortest path for all pairs from only one source node results in 140,969,120 observations, which is a set of size 11 GB. Finding shortest paths for all pairs from all nodes would produce an enormous set.

The SPPATHS= set contains the following tuple members:

1. the source node of this shortest path
2. the sink node of this shortest path
3. for this source-sink pair, the order of this link in a shortest path
4. the *tail* node of this link in a shortest path
5. the *head* node of this link in a shortest path

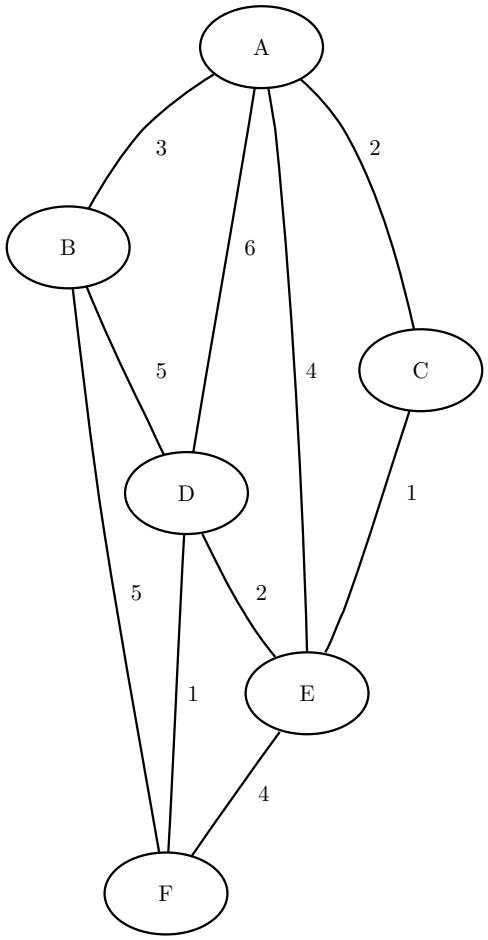
SPWEIGHTS= Array

This array contains the total weight for the shortest path for each of the source-sink pairs.

Shortest Paths for All Pairs

This example illustrates the use of the shortest path algorithm for all source-sink pairs on the simple undirected graph *G* that is shown in Figure 9.52.

Figure 9.52 A Simple Undirected Graph *G*



The undirected graph *G* can be represented by the following links data set, LinkSetIn:

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 3  A C 2  A D 6  A E 4  B D 5
B F 5  C E 1  D E 2  D F 1  E F 4
;
```

The following statements calculate shortest paths for all source-sink pairs:

```
proc optmodel;
   set <str,str> LINKS;
   num weight{LINKS};
   read data LinkSetIn into LINKS=[from to] weight;
```

```
   set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
   set NODES = union{<i,j> in LINKS} {i,j};
   num path_length{NODES, NODES};

   solve with NETWORK /
      links     = (weight=weight)
      shortpath
      out       = (sppaths=PATHS spweights=path_length)
   ;

   put PATHS;
   print path_length;
   create data ShortPathP from [source sink order from to]=PATHS
      weight[from,to];
   create data ShortPathW from [source sink]
      path_weight=path_length;
quit;
```

The data set ShortPathP contains the shortest paths and is shown in Figure 9.53.

Figure 9.53 All-Pairs Shortest Paths

ShortPathP

source	sink	order	from	to	weight
A	B	1	A	B	3
A	C	1	A	C	2
A	D	1	A	C	2
A	D	2	C	E	1
A	D	3	D	E	2
A	E	1	A	C	2
A	E	2	C	E	1
A	F	1	A	C	2
A	F	2	C	E	1
A	F	3	D	E	2
A	F	4	D	F	1
B	A	1	A	B	3
B	C	1	A	B	3
B	C	2	A	C	2
B	D	1	B	D	5
B	E	1	A	B	3
B	E	2	A	C	2
B	E	3	C	E	1
B	F	1	B	F	5
C	A	1	A	C	2
C	B	1	A	C	2
C	B	2	A	B	3
C	D	1	C	E	1
C	D	2	D	E	2
C	E	1	C	E	1
C	F	1	C	E	1
C	F	2	D	E	2
C	F	3	D	F	1
D	A	1	D	E	2
D	A	2	C	E	1
D	A	3	A	C	2
D	B	1	B	D	5
D	C	1	D	E	2
D	C	2	C	E	1
D	E	1	D	E	2
D	F	1	D	F	1
E	A	1	C	E	1
E	A	2	A	C	2
E	B	1	C	E	1
E	B	2	A	C	2
E	B	3	A	B	3
E	C	1	C	E	1
E	D	1	D	E	2
E	F	1	D	E	2
E	F	2	D	F	1
F	A	1	D	F	1
F	A	2	D	E	2
F	A	3	C	E	1
F	A	4	A	C	2
F	B	1	B	F	5
F	C	1	D	F	1
F	C	2	D	E	2
F	C	3	C	E	1
F	D	1	D	F	1
F	E	1	D	F	1
F	E	2	D	E	2

The data set ShortPathW contains the path weight for the shortest paths of each source-sink pair and is shown in Figure 9.54.

Figure 9.54 All-Pairs Shortest Paths Summary

ShortPathW

source	sink	path_weight
A	A	.
A	B	3
A	C	2
A	D	5
A	E	3
A	F	6
B	A	3
B	B	.
B	C	5
B	D	5
B	E	6
B	F	5
C	A	2
C	B	5
C	C	.
C	D	3
C	E	1
C	F	4
D	A	5
D	B	5
D	C	3
D	D	.
D	E	2
D	F	1
E	A	3
E	B	6
E	C	1
E	D	2
E	E	.
E	F	3
F	A	6
F	B	5
F	C	4
F	D	1
F	E	3
F	F	.

When you are interested only in the source-sink pair that has the longest shortest path, you can use the PATHS= suboption. This suboption affects only the output processing; it does not affect the computation. All the designated source-sink shortest paths are calculated, but only the longest ones are written to the output set.

The following statements display only the longest shortest paths:

```
proc optmodel;
   set <str,str> LINKS;
   num weight{LINKS};
   read data LinkSetIn into LINKS=[from to] weight;
   set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */

   solve with NETWORK /
      links     = ( weight = weight )
      shortpath = ( paths = longest )
      out       = ( sppaths = PATHS )
   ;

   put PATHS;
   create data ShortPathLong from [source sink order from to]=PATHS
      weight[from,to];
quit;
```

The data set ShortPathLong now contains the longest shortest paths and is shown in Figure 9.55.

Figure 9.55 Longest Shortest Paths

ShortPathLong

source	sink	order	from	to	weight
A	F	1	A	C	2
A	F	2	C	E	1
A	F	3	D	E	2
A	F	4	D	F	1
B	E	1	A	B	3
B	E	2	A	C	2
B	E	3	C	E	1
E	B	1	C	E	1
E	B	2	A	C	2
E	B	3	A	B	3
F	A	1	D	F	1
F	A	2	D	E	2
F	A	3	C	E	1
F	A	4	A	C	2

Shortest Paths for a Subset of Source-Sink Pairs

This section illustrates the use of the SOURCE= and SINK= suboptions and the shortest path algorithm to calculate shortest paths for a subset of source-sink pairs. If S denotes the nodes in the SOURCE= set and T denotes the nodes in the SINK= set, the network solver calculates all the source-sink pairs in the crossproduct of these two sets.

For example, the following statements calculate a shortest path for the four combinations of source-sink pairs in $S \times T = \{A, C\} \times \{B, F\}$:

```
proc optmodel;
   set <str,str> LINKS;
   num weight{LINKS};
   read data LinkSetIn into LINKS=[from to] weight;
   set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
   set SOURCES = / A C /;
   set SINKS   = / B F /;

   solve with NETWORK /
      links     = (weight=weight)
      shortpath = (source=SOURCES sink=SINKS)
      out       = (sppaths=PATHS)
   ;

   put PATHS;
   create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

The data set ShortPath contains the shortest paths and is shown in Figure 9.56.

Figure 9.56 Shortest Paths for a Subset of Source-Sink Pairs

ShortPath

source	sink	order	from	to	weight
A	B	1	A	B	3
A	F	1	A	C	2
A	F	2	C	E	1
A	F	3	D	E	2
A	F	4	D	F	1
C	B	1	A	C	2
C	B	2	A	B	3
C	F	1	C	E	1
C	F	2	D	E	2
C	F	3	D	F	1

Shortest Paths for a Subset of Source or Sink Pairs

This section illustrates the use of the shortest path algorithm to calculate shortest paths between a subset of source (or sink) nodes and all other sink (or source) nodes.

In this case, you designate the subset of source (or sink) nodes in the node set by specifying the SOURCE= (or SINK=) suboption. By specifying only one of the suboptions, you indicate that you want the network solver to calculate all pairs from a subset of source nodes (or to calculate all pairs to a subset of sink nodes).

For example, the following statements calculate all the shortest paths from nodes B and E.:

```
proc optmodel;
   set <str,str> LINKS;
   num weight{LINKS};
   read data LinkSetIn into LINKS=[from to] weight;
   set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
   set SOURCES = / B E /;

   solve with NETWORK /
      links     = (weight=weight)
      shortpath = (source=SOURCES)
      out       = (sppaths=PATHS)
   ;

   put PATHS;
   create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

The data set ShortPath contains the shortest paths and is shown in Figure 9.57.

Figure 9.57 Shortest Paths for a Subset of Source Pairs

ShortPath

source	sink	order	from	to	weight
B	A	1	A	B	3
B	C	1	A	B	3
B	C	2	A	C	2
B	D	1	B	D	5
B	E	1	A	B	3
B	E	2	A	C	2
B	E	3	C	E	1
B	F	1	B	F	5
E	A	1	C	E	1
E	A	2	A	C	2
E	B	1	C	E	1
E	B	2	A	C	2
E	B	3	A	B	3
E	C	1	C	E	1
E	D	1	D	E	2
E	F	1	D	E	2
E	F	2	D	F	1

Conversely, the following statements calculate all the shortest paths to nodes B and E.:

```
proc optmodel;
   set <str,str> LINKS;
   num weight{LINKS};
   read data LinkSetIn into LINKS=[from to] weight;
   set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
   set SINKS = / B E /;

   solve with NETWORK /
      links     = (weight=weight)
      shortpath = (sink=SINKS)
      out       = (sppaths=PATHS)
   ;

   put PATHS;
   create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

The data set ShortPath contains the shortest paths and is shown in Figure 9.58.

Figure 9.58 Shortest Paths for a Subset of Sink Pairs

ShortPath

source	sink	order	from	to	weight
A	B	1	A	B	3
A	E	1	A	C	2
A	E	2	C	E	1
B	E	1	A	B	3
B	E	2	A	C	2
B	E	3	C	E	1
C	B	1	A	C	2
C	B	2	A	B	3
C	E	1	C	E	1
D	B	1	B	D	5
D	E	1	D	E	2
E	B	1	C	E	1
E	B	2	A	C	2
E	B	3	A	B	3
F	B	1	B	F	5
F	E	1	D	F	1
F	E	2	D	E	2

Shortest Paths for One Source-Sink Pair

This section illustrates the use of the shortest path algorithm to calculate shortest paths between one source-sink pair by using the SOURCE= and SINK= suboptions.

The following statements calculate a shortest path between node *C* and node *F*:

```
proc optmodel;
   set <str,str> LINKS;
   num weight{LINKS};
   read data LinkSetIn into LINKS=[from to] weight;
   set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
   set SOURCES = / C /;
   set SINKS   = / F /;

   solve with NETWORK /
      links     = (weight=weight)
      shortpath = (source=SOURCES sink=SINKS)
      out       = (sppaths=PATHS)
   ;

   put PATHS;
   create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

The data set ShortPath contains this shortest path and is shown in Figure 9.59.

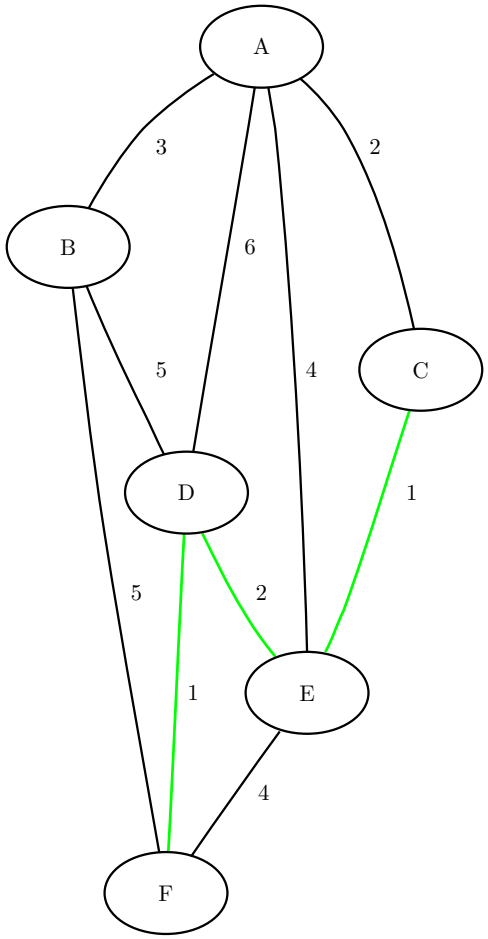
Figure 9.59 Shortest Paths for One Source-Sink Pair

ShortPath

source	sink	order	from	to	weight
C	F	1	C	E	1
C	F	2	D	E	2
C	F	3	D	F	1

The shortest path is shown graphically in Figure 9.60.

Figure 9.60 Shortest Path between Nodes C and F



Shortest Paths with Auxiliary Weight Calculation

This section illustrates the use of the shortest path algorithm, where auxiliary weights are used to calculate the shortest paths between all source-sink pairs.

Consider a links data set in which the auxiliary weight is a counter for each link:

```
data LinkSetIn;
   input from $ to $ weight count @@;
   datalines;
A B 3 1  A C 2 1  A D 6 1  A E 4 1  B D 5 1
B F 5 1  C E 1 1  D E 2 1  D F 1 1  E F 4 1
;
```

The following statements calculate shortest paths for all source-sink pairs:

```
proc optmodel;
   set <str,str> LINKS;
   num weight{LINKS};
   num count{LINKS};
   read data LinkSetIn into LINKS=[from to] weight count;
   set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
   set NODES = union{<i,j> in LINKS} {i,j};
   num path_length{i in NODES, j in NODES: i ~= j};

   solve with NETWORK /
      links     = (weight=weight)
      shortpath
      out       = (sppaths=PATHS spweights=path_length)
   ;

   put PATHS;
   num path_weight2{source in NODES, sink in NODES: source ~= sink} =
      sum {<(source),(sink),order,from,to> in PATHS} count[from,to];
   print path_length path_weight2;
   create data ShortPathW from [source sink]
      path_weight=path_length path_weight2;
quit;
```

The data set ShortPathW contains the total path weight for shortest paths in each source-sink pair and is shown in Figure 9.61. Because the variable count in LinkSetIn has a value of 1 for all links, the value in the output data set variable path_weights2 contains the number of links in each shortest path.

Figure 9.61 Shortest Paths Including Auxiliary Weights in Calculation

ShortPathW

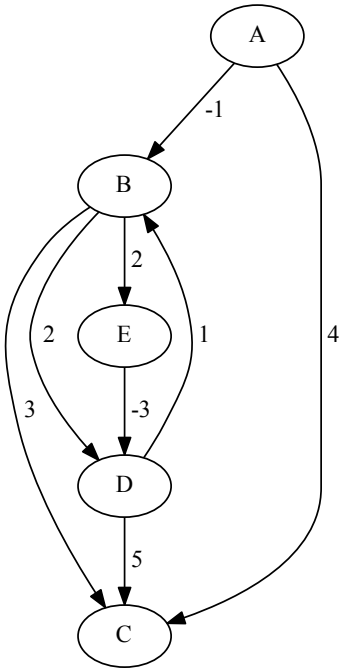
source	sink	path_weight	path_weight2
A	B	3	1
A	C	2	1
A	D	5	3
A	E	3	2
A	F	6	4
B	A	3	1
B	C	5	2
B	D	5	1
B	E	6	3
B	F	5	1
C	A	2	1
C	B	5	2
C	D	3	2
C	E	1	1
C	F	4	3
D	A	5	3
D	B	5	1
D	C	3	2
D	E	2	1
D	F	1	1
E	A	3	2
E	B	6	3
E	C	1	1
E	D	2	1
E	F	3	2
F	A	6	4
F	B	5	1
F	C	4	3
F	D	1	1
F	E	3	2

The section "Getting Started: Network Solver" on page 376 shows an example of using the shortest path algorithm to minimize travel to and from work based on traffic conditions.

Shortest Paths with Negative Link Weights

This section illustrates the use of the shortest path algorithm on a simple directed graph G with negative link weights, shown in Figure 9.62.

Figure 9.62 A Simple Directed Graph *G* with Negative Link Weights



The following statements call PROC OPTMODEL and declare the directed graph *G* by using set and array literals. For more information about literals, see the section "NUMBER, STRING, and SET Parameter Declarations" on page 47 in Chapter 5, "The OPTMODEL Procedure."

```
proc optmodel;
   set LINKS            = / <A B> <A C> <B C> <B D> <B E> <D B> <D C> <E D> /;
   num weight{LINKS} init [   -1     4     3     2     2     1     5    -3  ];
```

The next statements declare a set of the correct type for path output and calculate the shortest paths between source node *E* and sink node *B*:

```
   set NODES = union{<i,j> in LINKS} {i,j};
   /* Use the type (in this case, STRING) of NODES but leave PATHS empty */
   set PATHS init {NODES,NODES,/0/,NODES,NODES:0};
   set SOURCE = /E/, SINK = /B/;
   solve with NETWORK /
      links     = ( weight = weight )
      direction = directed
      shortpath = ( source = SOURCE sink = SINK )
      out       = ( sppaths = PATHS )
   ;
   put "Path and Weight: " (setof{<s,t,i,u,v> in PATHS} <u,v,weight[u,v]> );
```

As shown in Figure 9.63, the network solver identifies a shortest path that has negative weights.

Figure 9.63 SOLVE WITH NETWORK Log: Shortest Paths with Negative Link Weights

```
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 8.
NOTE: Processing the shortest paths problem.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
Path and Weight: {<'E','D',-3>,<'D','B',1>}
```

If you reduce the weight on link (B, E) from 2 units to 1 unit, there is a negative weight cycle ($E \rightarrow D \rightarrow B \rightarrow E$). The Bellman-Ford algorithm catches this and produces an error, as shown in Figure 9.64.

```
   weight['B','E'] = 1;
   solve with NETWORK /
      links     = (weight=weight)
      direction = directed
      shortpath = ( source = SOURCE sink = SINK )
      out       = ( sppaths = PATHS )
   ;
   put _SOLUTION_STATUS_=;
quit;
```

Figure 9.64 SOLVE WITH NETWORK Log: Negative Weight Cycle

```
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 8.
NOTE: Processing the shortest paths problem.
ERROR: The graph contains a negative weight cycle.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
_SOLUTION_STATUS_=BAD_PROBLEM_TYPE
```

Transitive Closure

The *transitive closure* of a graph G is a graph $G^T = (N, A^T)$ such that for all $i, j \in N$ there is a link $(i, j) \in A^T$ if and only if there exists a path from i to j in G.

The transitive closure of a graph can help you efficiently answer questions about reachability. Suppose you want to answer the question of whether you can get from node i to node j in the original graph G. Given the transitive closure G^T of G, you can simply check for the existence of link (i, j) to answer the question. Transitive closure has many applications, including speeding up the processing of structured query languages, which are often used in databases.

In the network solver, you can invoke the transitive closure algorithm by using the TRANSITIVE_CLOSURE option.

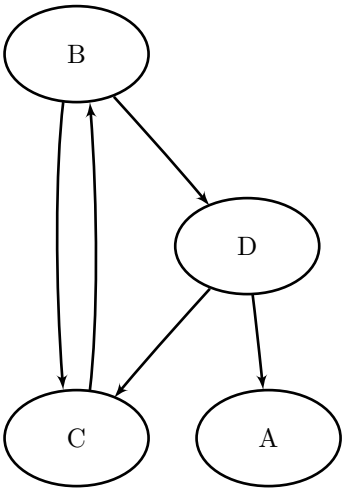
The results for the transitive closure algorithm are written to the set that is specified in the CLOSURE= suboption of the OUT= option.

The algorithm that the network solver uses to compute transitive closure is a sparse version of the Floyd-Warshall algorithm (Cormen, Leiserson, and Rivest 1990). This algorithm runs in time $O(|N|^3)$ and therefore might not scale to very large graphs.

Transitive Closure of a Simple Directed Graph

This example illustrates the use of the transitive closure algorithm on the simple directed graph *G* that is shown in Figure 9.65.

Figure 9.65 A Simple Directed Graph *G*



The directed graph *G* can be represented by the links data set LinkSetIn as follows:

```
data LinkSetIn;
   input from $ to $ @@;
   datalines;
B C  B D  C B  D A  D C
;
```

The following statements calculate the transitive closure and output the results in the data set TransClosure:

```
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetIn into LINKS=[from to];
   set<str,str> CAN_REACH;

   solve with NETWORK /
      links = ( include = LINKS )
      transc
      out = ( closure = CAN_REACH )
   ;

   put CAN_REACH;
   create data TransClosure from [from to]=CAN_REACH;
quit;
```

The data set TransClosure contains the transitive closure of G and is shown in Figure 9.66.

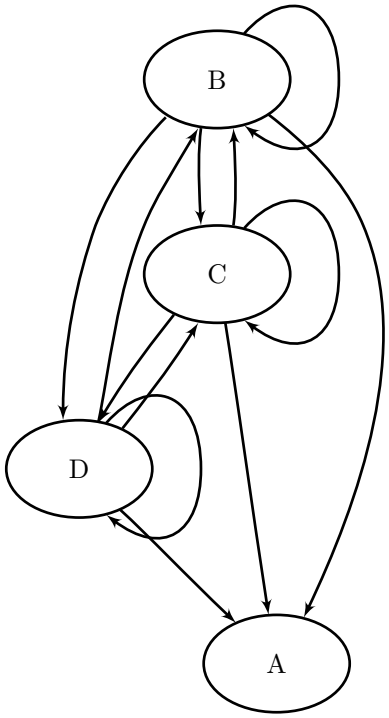
Figure 9.66 Transitive Closure of a Simple Directed Graph

Transitive Closure

from	to
B	C
B	D
D	A
D	C
C	C
D	D
B	B
B	A
C	A
A	A

The transitive closure of G is shown graphically in Figure 9.67.

Figure 9.67 Transitive Closure of G



For a more detailed example, see Example 9.6.

Traveling Salesman Problem

The *traveling salesman problem* (TSP) finds a minimum-cost tour in an undirected graph, G, that has a node set, N, and a link set, A. A *path* in a graph is a sequence of nodes, each of which has a link to the next node in the sequence. An *elementary cycle* is a path in which the start node and end node are the same and otherwise no node appears more than once in the sequence. A *Hamiltonian cycle* (or *tour*) is an elementary cycle that visits every node. In solving the TSP, then, the goal is to find a Hamiltonian cycle of minimum total cost, where the total cost is the sum of the costs of the links in the tour. Associated with each link $(i, j) \in A$ are a binary variable x_{ij}, which indicates whether link x_{ij} is part of the tour, and a cost c_{ij}. Let $\delta(S) = \{(i, j) \in A \mid i \in S,\ j \notin S\}$. Then an integer linear programming formulation of the TSP (for an undirected graph G) is as follows:

$$
\begin{array}{lllll}
\text{minimize} & \displaystyle\sum_{(i,j)\in A} c_{ij}x_{ij} & & & \\
\text{subject to} & \displaystyle\sum_{(i,j)\in\delta(i)} x_{i,j} = 2 & i \in N & & \text{(two\_match)} \\
& \displaystyle\sum_{(i,j)\in\delta(S)} x_{ij} \geq 2 & S \subset N,\ 2 \leq |S| \leq |N| - 1 & & \text{(subtour\_elim)} \\
& x_{ij} \in \{0, 1\} & (i, j) \in A & &
\end{array}
$$

The equations (two_match) are the *matching constraints*, which ensure that each node has degree two in the subgraph. The inequalities (subtour_elim) are the *subtour elimination constraints* (SECs), which enforce connectivity.

For a directed graph, G, the same formulation and solution approach is used on an expanded graph G', as described in Kumar and Li (1994). The network solver takes care of the construction of the expanded graph and returns the solution in terms of the original input graph.

In practical terms, you can think of the TSP in the context of a routing problem in which each node is a city and the links are roads that connect those cities. If you know the distance between each pair of cities, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city. The TSP has applications in planning, logistics, manufacturing, genomics, and many other areas.

In the network solver, you can invoke the traveling salesman problem solver by using the TSP= option.

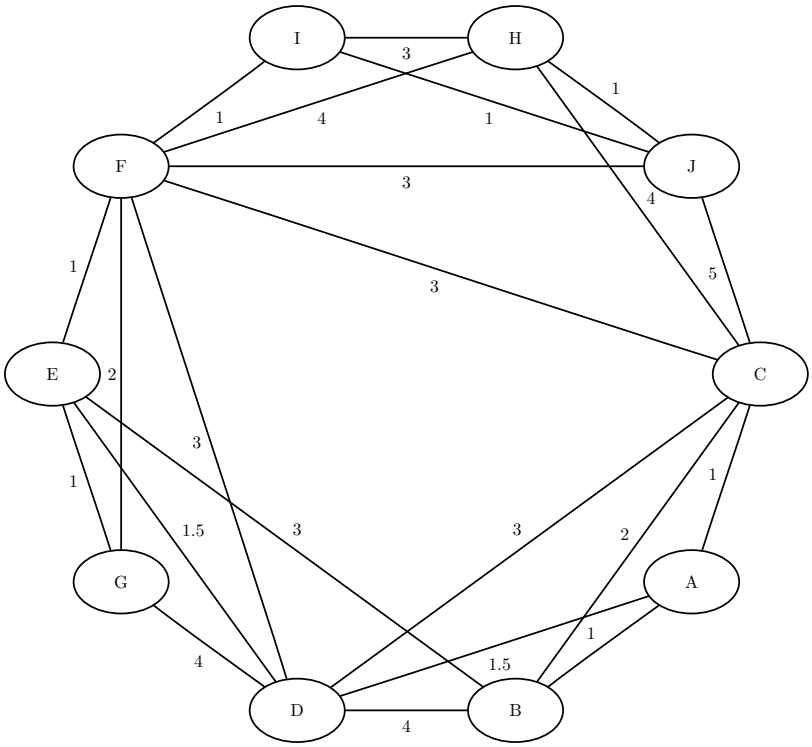
The algorithm that the network solver uses for solving a TSP is based on a variant of the branch-and-cut process described in Applegate et al. (2006).

The resulting tour is represented in two ways: in the numeric array that is specified in the ORDER= suboption in the SOLVE WITH NETWORK statement, the tour is specified as a sequence of nodes; in the set that is specified in the TOUR= suboption of the TSP option, the tour is specified as a list of links in the optimal tour.

Traveling Salesman Problem Applied to a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 9.68.

Figure 9.68 A Simple Undirected Graph



You can represent the links data set as follows:

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 1.0   A C 1.0   A D 1.5   B C 2.0   B D 4.0
B E 3.0   C D 3.0   C F 3.0   C H 4.0   D E 1.5
D F 3.0   D G 4.0   E F 1.0   E G 1.0   F G 2.0
F H 4.0   H I 3.0   I J 1.0   C J 5.0   F J 3.0
F I 1.0   H J 1.0
;
```

The following statements calculate an optimal traveling salesman tour and output the results in the data sets TSPTour and NodeSetOut:

```
proc optmodel;
   set<str,str> EDGES;
   set<str> NODES = union{<i,j> in EDGES} {i,j};
   num weight{EDGES};
   read data LinkSetIn into EDGES=[from to] weight;
   num tsp_order{NODES};
   set<str,str> TOUR;

   solve with NETWORK /
      loglevel = moderate
```

```
      links    = (weight=weight)
      tsp
      out      = (order=tsp_order tour=TOUR)
   ;

   put TOUR;
   print {<i,j> in TOUR} weight;
   print tsp_order;
   create data NodeSetOut from [node]         tsp_order;
   create data TSPTour    from [from to]=TOUR weight;
quit;
```

The progress of the procedure is shown in Figure 9.69.

Figure 9.69 Network Solver Log: Optimal Traveling Salesman Tour of a Simple Undirected Graph

```
NOTE: There were 22 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 22.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 16 using 0.00 (cpu:
      0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       1     16.0000000     15.5005000    3.22%       0
             0       0       1     16.0000000     16.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 16.
NOTE: Processing the traveling salesman problem used 0.01 (cpu: 0.00) seconds.
{<'A','B'>,<'B','C'>,<'C','H'>,<'H','J'>,<'I','J'>,<'F','I'>,<'F','G'>,<'E','G'>
,<'D','E'>,<'A','D'>}
NOTE: The data set WORK.NODESETOUT has 10 observations and 2 variables.
NOTE: The data set WORK.TSPTOUR has 10 observations and 3 variables.
```

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 9.70.

Figure 9.70 Nodes in the Optimal Traveling Salesman Tour

Traveling Salesman Problem

node	tsp_order
A	1
B	2
C	3
H	4
J	5
I	6
F	7
G	8
E	9
D	10

The data set TSPTour now contains the links in the optimal tour and is shown in Figure 9.71.

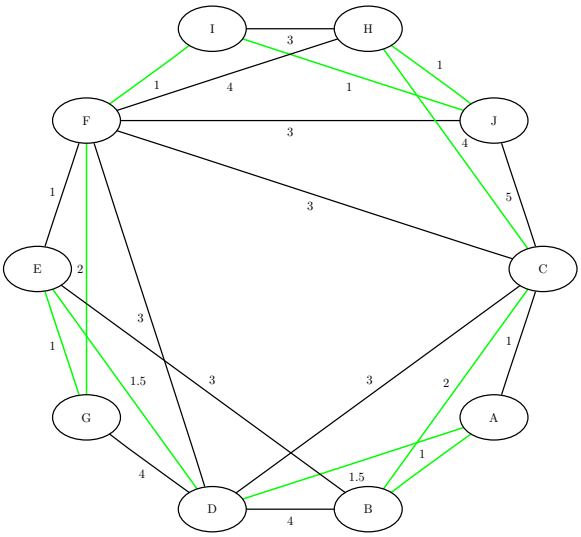
Figure 9.71 Links in the Optimal Traveling Salesman Tour

Traveling Salesman Problem

from	to	weight
A	B	1.0
B	C	2.0
C	H	4.0
H	J	1.0
I	J	1.0
F	I	1.0
F	G	2.0
E	G	1.0
D	E	1.5
A	D	1.5
		16.0

The minimum-cost links are shown in green in Figure 9.72.

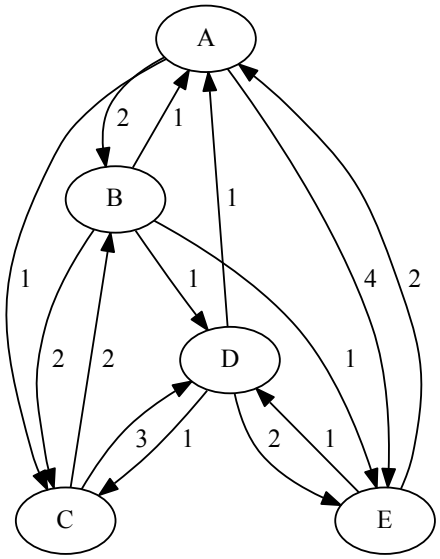
Figure 9.72 Optimal Traveling Salesman Tour



Traveling Salesman Problem Applied to a Simple Directed Graph

As another simple example, consider the weighted directed graph in Figure 9.73.

Figure 9.73 A Simple Directed Graph



You can represent the links data set as follows:

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 2.0   A C 1.0   A E 4.0
B A 1.0   B C 2.0   B D 1.0   B E 1.0
C B 2.0   C D 3.0
D A 1.0   D C 1.0   D E 2.0
E A 2.0   E D 1.0
;
```

The following statements, which are identical to those in the undirected example above except for the SOLVE statement clause DIRECTION=DIRECTED, calculate an optimal traveling salesman tour (on a directed graph) and output the results in the data sets TSPTour and NodeSetOut:

```
proc optmodel;
   set<str,str> EDGES;
   set<str> NODES = union{<i,j> in EDGES} {i,j};
   num weight{EDGES};
   read data LinkSetIn into EDGES=[from to] weight;
   num tsp_order{NODES};
   set<str,str> TOUR;

   solve with NETWORK /
      loglevel  = moderate
```

```
      links     = (weight=weight)
      direction = directed
      tsp
      out       = (order=tsp_order tour=TOUR)
   ;

   put TOUR;
   print {<i,j> in TOUR} weight;
   print tsp_order;
   create data NodeSetOut from [node]         tsp_order;
   create data TSPTour    from [from to]=TOUR weight;
quit;
```

The progress of the procedure is shown in Figure 9.74.

Figure 9.74 Network Solver Log: Optimal Traveling Salesman Tour of a Simple Directed Graph

```
NOTE: There were 14 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 14.
NOTE: The TSP solver is starting using an augmented symmetric graph with 10
      nodes and 19 links.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 6 using 0.00 (cpu:
      0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       1      6.0000000      5.9001000    1.69%       0
             0       0       1      6.0000000      6.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 6.
NOTE: Processing the traveling salesman problem used 0.01 (cpu: 0.00) seconds.
{<'A','C'>,<'C','B'>,<'B','E'>,<'E','D'>,<'D','A'>}
NOTE: The data set WORK.NODESETOUT has 5 observations and 2 variables.
NOTE: The data set WORK.TSPTOUR has 5 observations and 3 variables.
```

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 9.75.

Figure 9.75 Nodes in the Optimal Traveling Salesman Tour

node	tsp_order
A	1
C	2
B	3
E	4
D	5

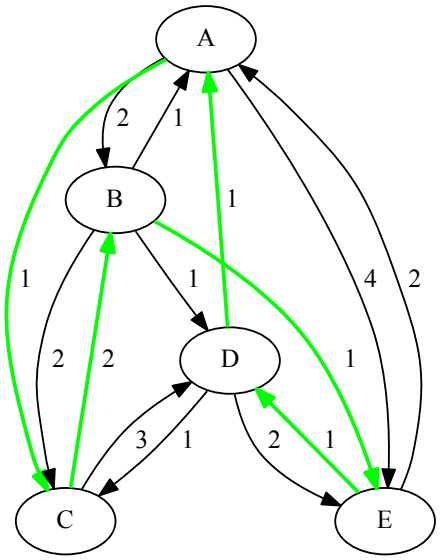
The data set TSPTour now contains the links in the optimal tour and is shown in Figure 9.76.

Figure 9.76 Links in the Optimal Traveling Salesman Tour

from	to	weight
A	C	1
C	B	2
B	E	1
E	D	1
D	A	1
		6

The minimum-cost links are shown in green in Figure 9.77.

Figure 9.77 Optimal Traveling Salesman Tour



Macro Variable _OROPTMODEL_

PROC OPTMODEL always creates and initializes a SAS macro variable called _OROPTMODEL_, which contains a character string. After each PROC OPTMODEL run, you can examine this macro variable by specifying `%put &_OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable.

Each keyword and value pair in _OROPTMODEL_ also appears in two other places: the PROC OPTMODEL automatic arrays _OROPTMODEL_NUM_ and _OROPTMODEL_STR_; and the ODS tables ProblemSummary and SolutionSummary, which appear after a SOLVE statement, unless you set the PRINTLEVEL= option to NONE. You can use these variables to obtain details about the solution even if you do not specify an output destination in the OUT= option.

After the solver is called, the various keywords in the variable are interpreted as follows:

STATUS
indicates the solver status at termination. It can take one of the following values:

OK	The solver terminated normally.
SYNTAX_ERROR	The use of syntax is incorrect.
DATA_ERROR	The input data is inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem in reading or writing of data has occurred.
SEMANTIC_ERROR	An evaluation error, such as an invalid operand type, has occurred.
ERROR	The status cannot be classified into any of the preceding categories.

SOLUTION_STATUS
indicates the solution status at termination. It can take one of the following values:

ABORT_NOSOL	The solver was stopped by the user and did not find a solution.
ABORT_SOL	The solver was stopped by the user but still found a solution.
BAD_PROBLEM_TYPE	The problem type is not supported by the solver.
CONDITIONAL_OPTIMAL	The optimality of the solution cannot be proven.
ERROR	The algorithm encountered an error.
FAIL_NOSOL	The solver stopped due to errors and did not find a solution.
FAIL_SOL	The solver stopped due to errors but still found a solution.
FAILED	The solver failed to converge, possibly due to numerical issues.
HEURISTIC_NOSOL	The solver used only heuristics and did not find a solution.
HEURISTIC_SOL	The solver used only heuristics and found a solution.
INFEASIBLE	The problem is infeasible.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
INTERRUPTED	The solver was interrupted by the system or the user before completing its work.
ITERATION_LIMIT_REACHED	The solver reached the maximum number of iterations that is specified in the MAXITER= option.
NODE_LIM_NOSOL	The solver reached the maximum number of nodes specified in the MAXNODES= option and did not find a solution.
NODE_LIM_SOL	The solver reached the maximum number of nodes specified in the MAXNODES= option and found a solution.

NULL_GRAPH	The graph was null (it had 0 nodes) after PROC OPTMODEL processed the SUBGRAPH= option.
OK	The algorithm terminated normally.
OPTIMAL	The solution is optimal.
OPTIMAL_AGAP	The solution is optimal within the absolute gap that is specified in the ABSOBJGAP= option.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances because of scaling.
OPTIMAL_RGAP	The solution is optimal within the relative gap that is specified in the RELOBJGAP= option.
OUTMEM_NOSOL	The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
OUTMEM_SOL	The solver ran out of memory but still found a solution.
SOLUTION_LIM	The solver reached the maximum number of solutions specified in the MAXCLIQUES=, MAXCYCLES=, or MAXSOLS= option.
TARGET	The solution is not worse than the target that is specified in the TARGET= option.
TIME_LIM_NOSOL	The solver reached the execution time limit specified in the MAXTIME= option and did not find a solution.
TIME_LIM_SOL	The solver reached the execution time limit specified in the MAXTIME= option and found a solution.
TIME_LIMIT_REACHED	The solver reached its execution time limit.
UNBOUNDED	The problem is unbounded.

PROBLEM_TYPE

indicates the type of problem solved. It can take one of the following values:

BICONCOMP	Biconnected components
CLIQUE	Maximal cliques
CONCOMP	Connected components
CYCLE	Cycle detection
LAP	Linear assignment (matching)
MCF	Minimum-cost network flow
MINCUT	Minimum cut
MST	Minimum spanning tree
SHORTPATH	Shortest path
TRANSCL	Transitive closure
TSP	Traveling salesman
NONE	This value is used when you do not specify an algorithm to run.

OBJECTIVE
indicates the objective value that is obtained by the solver at termination. For problem classes that do not have an explicit objective, such as cycle, the value of this keyword within the _OROPTMODEL_ macro variable is missing (.).

RELATIVE_GAP
indicates the relative gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the MILP solver. The relative gap is equal to

$$| \text{BestInteger} - \text{BestBound} | / (1\text{E}{-}10 + | \text{BestBound} |)$$

ABSOLUTE_GAP
indicates the absolute gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the MILP solver. The absolute gap is equal to $| \text{BestInteger} - \text{BestBound} |$.

PRIMAL_INFEASIBILITY
indicates the maximum (absolute) violation of the primal constraints by the solution.

BOUND_INFEASIBILITY
indicates the maximum (absolute) violation by the solution of the lower or upper bounds (or both).

INTEGER_INFEASIBILITY
indicates the maximum (absolute) violation of the integrality of integer variables returned by the MILP solver.

BEST_BOUND
indicates the best bound on the objective function value at termination. A missing value indicates that the MILP solver was not able to obtain such a bound.

NODES
indicates the number of nodes enumerated by the MILP solver by using the branch-and-bound algorithm.

ITERATIONS
indicates the number of simplex iterations taken to solve the problem.

PRESOLVE_TIME
indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

The following keywords within the _OROPTMODEL_ macro variable appear only with certain algorithms. The keywords convey information about the number of solutions each algorithm found:

NUM_ARTICULATION_POINTS
indicates the number of articulation points found. This term appears only for biconnected components.

NUM_CLIQUES
indicates the number of cliques found. This term appears only for clique.

NUM_COMPONENTS
indicates the number of components that match the definitions of the corresponding problem class. This term appears only for connected components and biconnected components.

NUM_CYCLES
indicates the number of cycles found that satisfy the criteria you provide. This term appears only for cycles.

Examples: Network Solver

Example 9.1: Articulation Points in a Terrorist Network

This example considers the terrorist communications network from the attacks on the United States on September 11, 2001, described in Krebs 2002. Figure 9.78 shows this network, which was constructed after the attacks, based on collected intelligence information.

Figure 9.78 Terrorist Communications Network from 9/11



The full network data include 153 links. The following statements show a small subset to illustrate the use of the BICONCOMP option in this context:

```
data LinkSetInTerror911;
   input from & $32. to & $32.;
   datalines;
Abu Zubeida                Djamal Beghal
Jean-Marc Grandvisir       Djamal Beghal
Nizar Trabelsi             Djamal Beghal
Abu Walid                  Djamal Beghal
Abu Qatada                 Djamal Beghal
Zacarias Moussaoui         Djamal Beghal
Jerome Courtaillier        Djamal Beghal
Kamel Daoudi               Djamal Beghal
Abu Walid                  Kamel Daoudi
Abu Walid                  Abu Qatada
Kamel Daoudi               Zacarias Moussaoui
Kamel Daoudi               Jerome Courtaillier
```

```
   ... more lines ...

Nawaf Alhazmi              Khalid Al-Mihdhar
Osama Awadallah            Khalid Al-Mihdhar
Abdussattar Shaikh         Khalid Al-Mihdhar
Abdussattar Shaikh         Osama Awadallah
;
```

Suppose that this communications network had been discovered before the attack on 9/11. If the investigators' goal was to disrupt the flow of communication between different groups within the organization, then they would want to focus on the people who are articulation points in the network.

To find the articulation points, use the following statements:

```
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetInTerror911 into LINKS=[from to];
   set NODES = union{<i,j> in LINKS} {i,j};
   set<str> ARTPOINTS;

   solve with NETWORK /
      links     = (include=LINKS)
      biconcomp
      out       = (artpoints=ARTPOINTS)
   ;

   put ARTPOINTS;
   create data ArtPoints from [node]=ARTPOINTS artpoint=1;
quit;
```

The data set ArtPoints contains members of the network who are articulation points. Focusing investigations on cutting off these particular members could have caused a great deal of disruption in the terrorists' ability to communicate when formulating the attack.

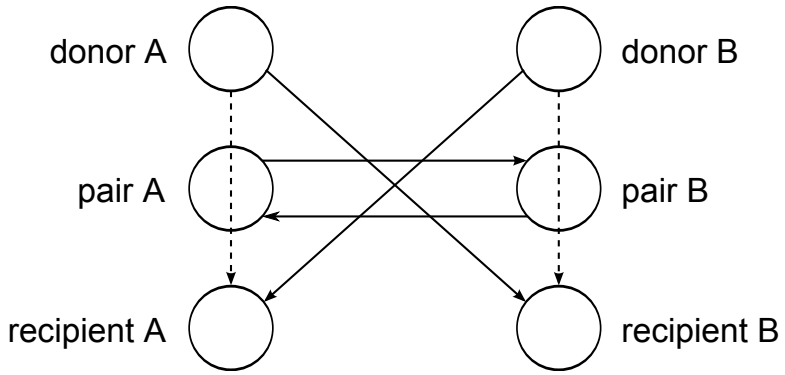
Output 9.1.1 Articulation Points of Terrorist Communications Network from 9/11

node	artpoint
Djamal Beghal	1
Zacarias Moussaoui	1
Essid Sami Ben Khemais	1
Mohamed Atta	1
Mamoun Darkazanli	1
Nawaf Alhazmi	1

Example 9.2: Cycle Detection for Kidney Donor Exchange

This example looks at an application of cycle detection to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to donate one. If the donor and recipient are incompatible (because of blood types, tissue mismatch, and so on), the transplant cannot happen. Now suppose two donor-recipient pairs A and B are in this situation, but donor A is compatible with recipient B and donor B is compatible with recipient A. Then two transplants can take place in a two-way swap, shown graphically in Figure 9.79. More generally, an n-way swap can be performed involving n donors and n recipients (Willingham 2009).

Figure 9.79 Kidney Donor Exchange Two-Way Swap



To model this problem, define a directed graph as follows. Each node is an incompatible donor-recipient pair. Link (i, j) exists if the donor from node i is compatible with the recipient from node j. The link weight is a measure of the quality of the match. By introducing dummy links whose weight is 0, you can also include altruistic donors who have no recipients or recipients who have no donors. The idea is to find a maximum-weight node-disjoint union of directed cycles. You want the union to be node-disjoint so that no kidney is donated more than once, and you want cycles so that the donor from node i gives up a kidney if and only if the recipient from node i receives a kidney.

Without any other constraints, the problem could be solved as a linear assignment problem, as described in the section "Linear Assignment (Matching)" on page 423. But doing so would allow arbitrarily long cycles in the solution. Because of practical considerations (such as travel) and to mitigate risk, each cycle must have no more than L links. The kidney exchange problem is to find a maximum-weight node-disjoint union of short directed cycles.

One way to solve this problem is to explicitly generate all cycles whose length is at most L and then solve a set packing problem. You can use PROC OPTMODEL to generate the cycles, formulate the set packing problem, call the mixed integer linear programming solver, and output the optimal solution.

The following DATA step sets up the problem, first creating a random graph on n nodes with link probability p and Uniform(0,1) weight:

```
/* create random graph on n nodes with arc probability p
   and uniform(0,1) weight */
%let n = 100;
%let p = 0.02;
data LinkSetIn;
   do from = 0 to &n - 1;
      do to = 0 to &n - 1;
         if from eq to then continue;
         else if ranuni(1) < &p then do;
            weight = ranuni(2);
            output;
         end;
      end;
   end;
run;
```

The following statements declare parameters and then read the input data:

```
%let max_length = 10;
proc optmodel;
   /* declare index sets and parameters, and read data */
   set <num,num> ARCS;
   num weight {ARCS};
   read data LinkSetIn into ARCS=[from to] weight;
   set<num,num,num> ID_ORDER_NODE;
```

The following statements use the network solver to generate all cycles whose length is greater than or equal to 2 and less than or equal to 10:

```
   /* generate all cycles with 2 <= length <= max_length */
   solve with NETWORK /
      loglevel        = moderate
      graph_direction = directed
      links           = (include=ARCS)
      cycle           = (mode=all_cycles minlength=2 maxlength=&max_length)
      out             = (cycles=ID_ORDER_NODE)
   ;
```

The network solver finds 224 cycles of the appropriate length, as shown in Output 9.2.1.

Output 9.2.1 Cycles for Kidney Donor Exchange Network Solver Log

```
NOTE: There were 194 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 97.
NOTE: The number of links in the input graph is 194.
NOTE: Processing cycle detection.
NOTE: The graph has 224 cycles.
NOTE: Processing cycle detection used 0.31 (cpu: 0.31) seconds.
```

From the resulting set ID_ORDER_NODE, use the following statements to convert to one tuple per cycle-arc combination:

```
   /* extract <cid,from,to> triples from <cid,order,node> triples */
   set <num,num,num> ID_FROM_TO init {};
   num last init ., from, to;
   for {<cid,order,node> in ID_ORDER_NODE} do;
      from = last;
      to   = node;
      last = to;
      if order ne 1 then ID_FROM_TO = ID_FROM_TO union {<cid,from,to>};
   end;
```

Given the set of cycles, you can now formulate a mixed integer linear program (MILP) to maximize the total cycle weight. Let C be the set of cycles of appropriate length, N_c be the set of nodes in cycle c, A_c be the set of links in cycle c, and w_{ij} be the link weight for link (i, j). Define a binary decision variable x_c. Set x_c to 1 if cycle c is used in the solution; otherwise, set it to 0. Then, the following MILP defines the problem that you want to solve (to maximize the quality of the kidney exchange):

$$\begin{aligned}
&\text{maximize} && \sum_{c \in C} \left(\sum_{(i,j) \in A_c} w_{ij} \right) x_c && && \\
&\text{subject to} && \sum_{c \in C : i \in N_c} x_c \leq 1 && i \in N && \text{(incomp_pair)} \\
& && x_c \in \{0, 1\} && c \in C &&
\end{aligned}$$

The constraint (incomp_pair) ensures that each node (incompatible pair) in the graph is intersected at most once. That is, a donor can donate a kidney only once. You can use PROC OPTMODEL to solve this mixed integer linear programming problem as follows:

```
   /* solve set packing problem to find maximum weight node-disjoint union
      of short directed cycles */
   set CYCLES = setof {<c,i,j> in ID_FROM_TO} c;
   set ARCS_c {c in CYCLES} = setof {<(c),i,j> in ID_FROM_TO} <i,j>;
   set NODES_c {c in CYCLES} = union {<i,j> in ARCS_c[c]} {i,j};
   set NODES = union {c in CYCLES} NODES_c[c];
   num cycle_weight {c in CYCLES} = sum {<i,j> in ARCS_c[c]} weight[i,j];

   /* UseCycle[c] = 1 if cycle c is used, 0 otherwise */
   var UseCycle {CYCLES} binary;

   /* declare objective */
   max TotalWeight
      = sum {c in CYCLES} cycle_weight[c] * UseCycle[c];

   /* each node appears in at most one cycle */
   con node_packing {i in NODES}:
      sum {c in CYCLES: i in NODES_c[c]} UseCycle[c] <= 1;

   /* call solver */
   solve with milp;

   /* output optimal solution */
   create data Solution from [c]={c in CYCLES: UseCycle[c].sol > 0.5}
      cycle_weight;
quit;
%put &_OROPTMODEL_;
```

PROC OPTMODEL solves the problem by using the mixed integer linear programming solver. As shown in Output 9.2.2, it was able to find a total weight (quality level) of 26.02.

Output 9.2.2 Cycles for Kidney Donor Exchange MILP Solver Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 224 variables (0 free, 0 fixed).
NOTE: The problem has 224 binary and 0 integer variables.
NOTE: The problem has 63 linear constraints (63 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 1900 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 46 variables and 35 constraints.
NOTE: The MILP presolver removed 901 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 178 variables, 28 constraints, and 999
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       3     22.7780692    868.9019355   97.38%       0
             0       1       3     22.7780692     26.7803921   14.94%       0
             0       1       4     23.2747070     26.0966646   10.81%       0
             0       1       4     23.2747070     26.0202906   10.55%       0
             0       1       5     26.0202875     26.0202875    0.00%       0
NOTE: The MILP solver added 8 cuts with 501 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 26.020287528.
NOTE: The data set WORK.SOLUTION has 6 observations and 2 variables.
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=26.020287528
RELATIVE_GAP=0 ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=1.110223E-15
BOUND_INFEASIBILITY=6E-7 INTEGER_INFEASIBILITY=2.8E-6 BEST_BOUND=26.020287528
NODES=1 ITERATIONS=142 PRESOLVE_TIME=0.00 SOLUTION_TIME=0.01
```

The data set Solution, shown in Output 9.2.3, now contains the cycles that define the best exchange and their associated weight (quality).

Output 9.2.3 Maximum Quality Solution for Kidney Donor Exchange

c	cycle_weight
12	5.84985
43	3.90015
71	5.44467
124	7.42574
222	2.28231
224	1.11757

Example 9.3: Linear Assignment Problem for Minimizing Swim Times

A swimming coach needs to assign male and female swimmers to each stroke of a medley relay team. The swimmers' best times for each stroke are stored in a SAS data set. The LINEAR_ASSIGNMENT option evaluates the times and matches strokes and swimmers to minimize the total relay swim time.

The data are stored in matrix format, where the row identifier is the swimmer's name (variable name) and each swimming event is a column (variables: back, breast, fly, and free). In the following DATA step, the relay times are split into two categories, male (M) and female (F):

```
data RelayTimes;
   input name $ sex $ back breast fly free;
   datalines;
Sue     F 35.1 36.7 28.3 36.1
Karen   F 34.6 32.6 26.9 26.2
Jan     F 31.3 33.9 27.1 31.2
Andrea  F 28.6 34.1 29.1 30.3
Carol   F 32.9 32.2 26.6 24.0
Ellen   F 27.8 32.5 27.8 27.0
Jim     M 26.3 27.6 23.5 22.4
Mike    M 29.0 24.0 27.9 25.4
Sam     M 27.2 33.8 25.2 24.1
Clayton M 27.0 29.2 23.0 21.9
;
```

The following statements solve the linear assignment problem for both male and female relay teams:

```
proc contents data=RelayTimes
   out=stroke_data(rename=(name=stroke) where=(type=1));
run;

proc optmodel;
   set <str> STROKES;
   read data stroke_data into STROKES=[stroke];
   set <str> SWIMMERS;
   str sex {SWIMMERS};
   num time {SWIMMERS, STROKES};
   read data RelayTimes into SWIMMERS=[name] sex
      {stroke in STROKES} <time[name,stroke]=col(stroke)>;
   set FEMALES = {i in SWIMMERS: sex[i] = 'F'};
   set FNODES = FEMALES union STROKES;
   set MALES = {i in SWIMMERS: sex[i] = 'M'};
   set MNODES = MALES union STROKES;
   set <str,str> PAIRS;

   solve with NETWORK /
      graph_direction = directed
      links           = (weight=time)
      subgraph        = (nodes=FNODES)
      lap
      out             = (assignments=PAIRS)
   ;
   put PAIRS;
   create data LinearAssignF from [name assign]=PAIRS sex[name] cost=time;

   solve with NETWORK /
      graph_direction = directed
      links           = (weight=time)
      subgraph        = (nodes=MNODES)
      lap
      out             = (assignments=PAIRS)
   ;
   put PAIRS;
   create data LinearAssignM from [name assign]=PAIRS sex[name] cost=time;
quit;
```

The progress of the two SOLVE WITH NETWORK calls is shown in Output 9.3.1.

Output 9.3.1 Network Solver Log: Linear Assignment for Swim Times

```
NOTE: The data set WORK.STROKE_DATA has 4 observations and 41 variables.
NOTE: There were 4 observations read from the data set WORK.STROKE_DATA.
NOTE: There were 10 observations read from the data set WORK.RELAYTIMES.
NOTE: The SUBGRAPH= option filtered 16 elements from 'time.'
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 24.
NOTE: Processing the linear assignment problem.
NOTE: Objective = 111.5.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
{<'Karen','breast'>,<'Jan','fly'>,<'Carol','free'>,<'Ellen','back'>}
NOTE: The data set WORK.LINEARASSIGNF has 4 observations and 4 variables.
NOTE: The SUBGRAPH= option filtered 24 elements from 'time.'
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 16.
NOTE: Processing the linear assignment problem.
NOTE: Objective = 96.6.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
{<'Jim','free'>,<'Mike','breast'>,<'Sam','back'>,<'Clayton','fly'>}
NOTE: The data set WORK.LINEARASSIGNM has 4 observations and 4 variables.
```

The data sets LinearAssignF and LinearAssignM contain the optimal assignments. Note that in the case of the female data, there are more people (set *S*) than there are strokes (set *T*). Therefore, the solver allows for some members of *S* to remain unassigned.

Output 9.3.2 Optimal Assignments for Best Female Swim Times

name	assign	sex	cost
Karen	breast	F	32.6
Jan	fly	F	27.1
Carol	free	F	24.0
Ellen	back	F	27.8
			111.5

Output 9.3.3 Optimal Assignments for Best Male Swim Times

name	assign	sex	cost
Jim	free	M	22.4
Mike	breast	M	24.0
Sam	back	M	27.2
Clayton	fly	M	23.0
			96.6

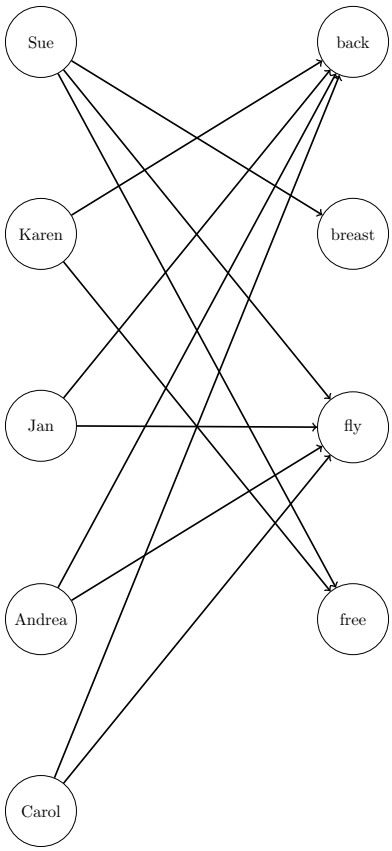
Example 9.4: Linear Assignment Problem, Sparse Format versus Dense Format

This example looks at the problem of assigning swimmers to strokes based on their best times. However, in this case certain swimmers are not eligible to perform certain strokes. A missing (.) value in the data matrix identifies an ineligible assignment. For example:

```
data RelayTimesMatrix;
   input name $ sex $ back breast fly free;
   datalines;
Sue     F   .  36.7 28.3 36.1
Karen   F 34.6    .    . 26.2
Jan     F 31.3    . 27.1    .
Andrea  F 28.6    . 29.1    .
Carol   F 32.9    . 26.6    .
;
```

Recall that the linear assignment problem can also be interpreted as the minimum-weight matching in a bipartite directed graph. The eligible assignments define links between the rows (swimmers) and the columns (strokes), as in Figure 9.80.

Figure 9.80 Bipartite Graph for Linear Assignment Problem



You can represent the same data in RelayTimesMatrix by using a links data set as follows:

```
data RelayTimesLinks;
   input name $ attr $ cost;
   datalines;
Sue     breast 36.7
Sue     fly    28.3
Sue     free   36.1
Karen   back   34.6
Karen   free   26.2
Jan     back   31.3
Jan     fly    27.1
Andrea  back   28.6
Andrea  fly    29.1
Carol   back   32.9
Carol   fly    26.6
;
```

This graph must be bipartite (such that *S* and *T* are disjoint). If it is not, the network solver returns an error.

Now, you can use either input format to solve the same problem, as follows:

```
proc contents data=RelayTimesMatrix
   out=stroke_data(rename=(name=stroke) where=(type=1));
run;

proc optmodel;
   set <str> STROKES;
   read data stroke_data into STROKES=[stroke];
   set <str> SWIMMERS;
   str sex {SWIMMERS};
   num time {SWIMMERS, STROKES};
   read data RelayTimesMatrix into SWIMMERS=[name]
      sex
      {stroke in STROKES} <time[name,stroke]=col(stroke)>;
   set SWIMMERS_STROKES =
      {name in SWIMMERS, stroke in STROKES: time[name,stroke] ne .};
   set <str,str> PAIRS;

   solve with NETWORK /
      graph_direction = directed
      links           = (weight=time)
      subgraph        = (links=SWIMMERS_STROKES)
      lap
      out             = (assignments=PAIRS)
   ;

   put PAIRS;
   create data LinearAssignMatrix from [name assign]=PAIRS
      sex[name] cost=time;
quit;

proc sql;
   create table stroke_data as
   select distinct attr as stroke
```

```
   from RelayTimesLinks;
quit;

proc optmodel;
   set <str> STROKES;
   read data stroke_data into STROKES=[stroke];
   set <str> SWIMMERS;
   str sex {SWIMMERS};
   set <str,str> SWIMMERS_STROKES;
   num time {SWIMMERS_STROKES};
   read data RelayTimesLinks into SWIMMERS_STROKES=[name attr] time=cost;
   set <str,str> PAIRS;

   solve with NETWORK /
      graph_direction = directed
      links           = (weight=time)
      lap
      out             = (assignments=PAIRS)
   ;

   put PAIRS;
   create data LinearAssignLinks from [name attr]=PAIRS cost=time;
quit;
```

The data sets LinearAssignMatrix and LinearAssignLinks now contain the optimal assignments, as shown in Output 9.4.1 and Output 9.4.2.

Output 9.4.1 Optimal Assignments for Swim Times (Dense Input)

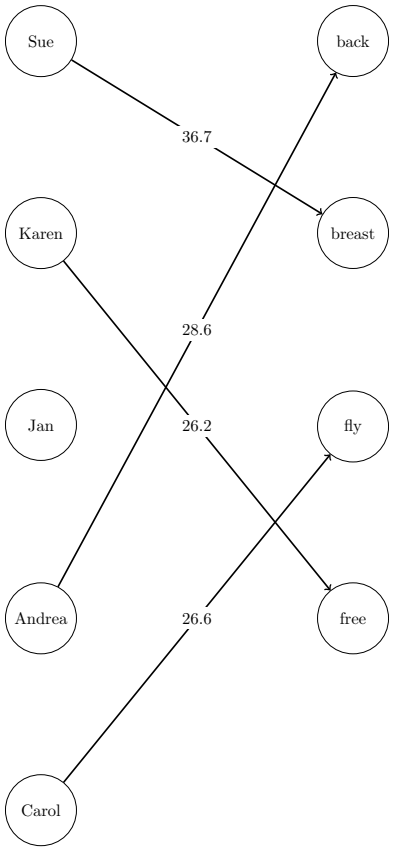
name	assign	sex	cost
Sue	breast	F	36.7
Karen	free	F	26.2
Andrea	back	F	28.6
Carol	fly	F	26.6
			118.1

Output 9.4.2 Optimal Assignments for Swim Times (Sparse Input)

name	attr	cost
Sue	breast	36.7
Karen	free	26.2
Andrea	back	28.6
Carol	fly	26.6
		118.1

The optimal assignments are shown graphically in Figure 9.81.

Figure 9.81 Optimal Assignments for Swim Times

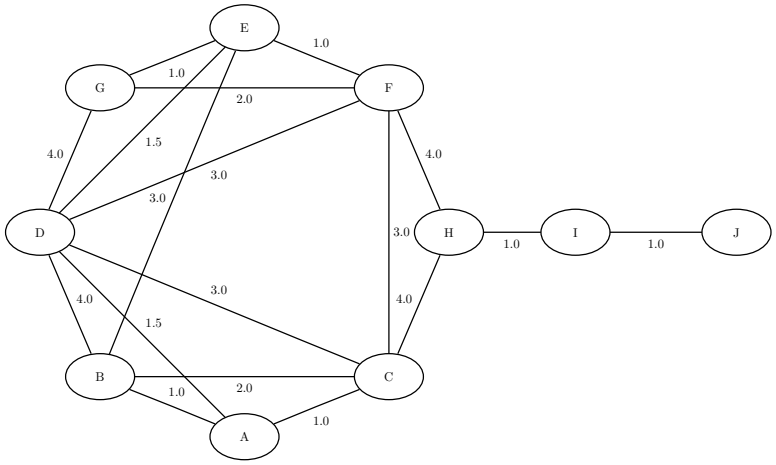


For large problems where a number of links are forbidden, the sparse format can be faster and can save a great deal of memory. Consider an example that uses the dense format with 15,000 columns ($|S| = 15,000$) and 4,000 rows ($|T| = 4,000$). To store the dense matrix in memory, the network solver needs to allocate approximately $|S| \cdot |T| \cdot 8/1024/1024 = 457$ MB. If the data have mostly ineligible links, then the sparse (graph) format is much more efficient with respect to memory. For example, if the data have only 5% of the eligible links ($15,000 \cdot 4,000 \cdot 0.05 = 3,000,000$), then the dense storage would still need 457 MB. The sparse storage for the same example needs approximately $|S| \cdot |T| \cdot 0.05 \cdot 12/1024/1024 = 34$ MB. If the problem is fully dense (all links are eligible), then the dense format is more efficient.

Example 9.5: Minimum Spanning Tree for Computer Network Topology

Consider the problem of designing a small network of computers in an office. In designing the network, the goal is to make sure that each machine in the office can reach every other machine. To accomplish this goal, Ethernet lines must be constructed and run between the machines. The construction costs for each possible link are based approximately on distance and are shown in Figure 9.82. Besides distance, the costs also reflect some restrictions due to physical boundaries. To connect all the machines in the office at minimal cost, you need to find a minimum spanning tree on the network of possible links.

Figure 9.82 Potential Office Computer Network



Define the link data set as follows:

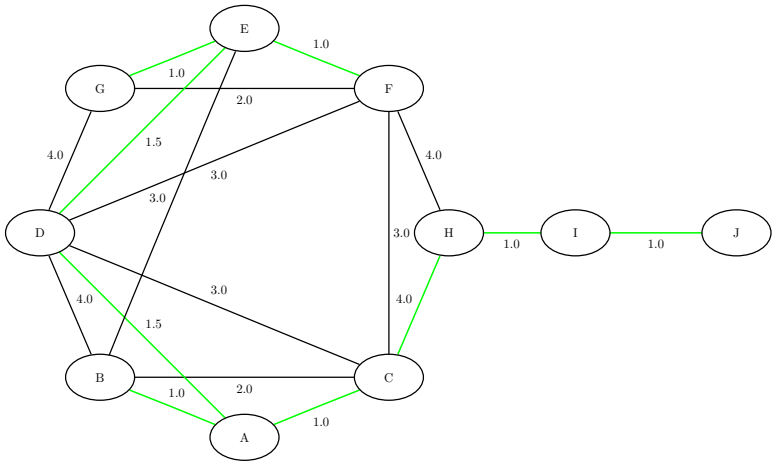
```
data LinkSetInCompNet;
   input from $ to $ weight @@;
   datalines;
A B 1.0  A C 1.0  A D 1.5  B C 2.0  B D 4.0
B E 3.0  C D 3.0  C F 3.0  C H 4.0  D E 1.5
D F 3.0  D G 4.0  E F 1.0  E G 1.0  F G 2.0
F H 4.0  H I 1.0  I J 1.0
;
```

The following statements find a minimum spanning tree:

```
proc optmodel;
   set<str,str> LINKS;
   num weight{LINKS};
   read data LinkSetInCompNet into LINKS=[from to] weight;
   set<str,str> FOREST;

   solve with NETWORK /
      links       = (weight=weight)
      minspantree
      out         = (forest=FOREST)
   ;

   put FOREST;
   put (sum {<i,j> in FOREST} weight[i,j]);
   create data MinSpanTree from [from to]=FOREST weight;
quit;
```

Output 9.5.1 shows the resulting data set MinSpanTree, which is displayed graphically in Figure 9.83 with the minimal cost links shown in green.

Figure 9.83 Minimum Spanning Tree for Office Computer Network



Output 9.5.1 Minimum Spanning Tree of a Computer Network Topology

from	to	weight
I	J	1.0
A	C	1.0
E	F	1.0
E	G	1.0
H	I	1.0
A	B	1.0
D	E	1.5
A	D	1.5
C	H	4.0
		13.0

Example 9.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

Most software bug tracking systems have some notion of *duplicate bugs* in which one bug is declared to be the same as another bug. If bug A is considered a duplicate (DUP) of bug B, then a fix for B would also fix A. You can represent the DUPs in a bug tracking system as a directed graph where you add a link $A \rightarrow B$ if A is a DUP of B.

The bug tracking system needs to check for two situations when users declare a bug to be a DUP. The first situation is called a *circular dependence*. Consider bugs A, B, C, and D in the tracking system. The first user declares that A is a DUP of B and that C is a DUP of D. Then, a second user declares that B is a DUP of C, and a third user declares that D is a DUP of A. You now have a circular dependence, and no primary bug is defined on which the development team should focus. You can easily see this circular dependence in the graph representation, because $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. Finding such circular dependencies can be done using cycle detection, which is described in the section "Cycle" on page 418. However, the second situation

that needs to be checked is more general. If a user declares that A is a DUP of B and another user declares that B is a DUP of C, this chain of duplicates is already an issue. The bug tracking system needs to provide one primary bug to which the rest of the bugs are duplicated. The existence of these chains can be identified by calculating the transitive closure of the directed graph that is defined by the DUP links.

Given the original directed graph G (defined by the DUP links) and its transitive closure G^T, any link in G^T that is not in G exists because of some chain that is present in G.

Consider the following data that define some duplicated bugs (called *defects*) in a small sample of the bug tracking system:

```
data DefectLinks;
   input defectId $ linkedDefect $ linkType $ when datetime16.;
   format when datetime16.;
   datalines;
D0096978 S0711218 DUPTO 20OCT10:00:00:00
S0152674 S0153280 DUPTO 30MAY02:00:00:00
S0153280 S0153307 DUPTO 30MAY02:00:00:00
S0153307 S0152674 DUPTO 30MAY02:00:00:00
S0162973 S0162978 DUPTO 29NOV10:16:13:16
S0162978 S0165405 DUPTO 29NOV10:16:13:16
S0325026 S0575748 DUPTO 01JUN10:00:00:00
S0347945 S0346582 DUPTO 03MAR06:00:00:00
S0350596 S0346582 DUPTO 21MAR06:00:00:00
S0539744 S0643230 DUPTO 10MAY10:00:00:00
S0575748 S0643230 DUPTO 15JUN10:00:00:00
S0629984 S0643230 DUPTO 01JUN10:00:00:00
;
```

The following statements calculate cycles in addition to the transitive closure of the graph G that is defined by the duplicated defects in DefectLinks. The output data set Cycles contains any circular dependencies, and the data set TransClosure contains the transitive closure G^T. To identify the chains, you can use PROC SQL to identify the links in G^T that are not in G.

```
proc optmodel;
   set<str,str> LINKS;
   read data DefectLinks into LINKS=[defectId linkedDefect];
   set<num,num,str> CYCLES;
   set<str,str> CLOSURE;

   solve with NETWORK /
      loglevel        = moderate
      graph_direction = directed
      links           = (include=LINKS)
      cycle           = (mode=first_cycle)
      out             = (cycles=CYCLES)
   ;

   put CYCLES;
   create data Cycles from [cycle order node]=CYCLES;
```

```
   solve with NETWORK /
      loglevel           = moderate
      graph_direction    = directed
      links              = (include=LINKS)
      transitive_closure
      out                = (closure=CLOSURE)
   ;

   put CLOSURE;
   create data TransClosure from [defectId linkedDefect]=CLOSURE;
quit;

proc sql;
   create table Chains as
   select defectId, linkedDefect from TransClosure
      except
   select defectId, linkedDefect from DefectLinks;
quit;
```

The progress of the procedure is shown in Output 9.6.1.

Output 9.6.1 Network Solver Log: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

```
NOTE: There were 12 observations read from the data set WORK.DEFECTLINKS.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: Processing cycle detection.
NOTE: The graph does have a cycle.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
{<1,1,'S0152674'>,<1,2,'S0153280'>,<1,3,'S0153307'>,<1,4,'S0152674'>}
NOTE: The data set WORK.CYCLES has 4 observations and 3 variables.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: Processing the transitive closure.
NOTE: Processing the transitive closure used 0.00 (cpu: 0.00) seconds.
{<'D0096978','S0711218'>,<'S0152674','S0153280'>,<'S0153280','S0153307'>,<
'S0153307','S0152674'>,<'S0162973','S0162978'>,<'S0162978','S0165405'>,<
'S0325026','S0575748'>,<'S0347945','S0346582'>,<'S0350596','S0346582'>,<
'S0539744','S0643230'>,<'S0575748','S0643230'>,<'S0629984','S0643230'>,<
'S0153307','S0153280'>,<'S0152674','S0153307'>,<'S0153307','S0153307'>,<
'S0152674','S0152674'>,<'S0153280','S0152674'>,<'S0153280','S0153280'>,<
'S0162973','S0165405'>,<'S0325026','S0643230'>}
NOTE: The data set WORK.TRANSCLOSURE has 20 observations and 2 variables.
NOTE: Table WORK.CHAINS created, with 8 rows and 2 columns.
```

The data set Cycles contains one case of a circular dependence in which the DUPs start and end at S0152674.

Output 9.6.2 Cycle in Bug Tracking System

cycle	order	node
1	1	S0152674
1	2	S0153280
1	3	S0153307
1	4	S0152674

The data set Chains contains the chains in the bug tracking system that come from the links in G^T that are not in G.

Output 9.6.3 Chains in Bug Tracking System

defectId	linkedDefect
S0152674	S0152674
S0152674	S0153307
S0153280	S0152674
S0153280	S0153280
S0153307	S0153280
S0153307	S0153307
S0162973	S0165405
S0325026	S0643230

Example 9.7: Traveling Salesman Tour through US Capital Cities

Consider a cross-country trip where you want to travel the fewest miles to visit all of the capital cities in all US states except Alaska and Hawaii. Finding the optimal route is an instance of the traveling salesman problem, which is described in section "Traveling Salesman Problem" on page 452.

The following PROC SQL statements use the built-in data set maps.uscity to generate a list of the capital cities and their latitude and longitude:

```
/* Get a list of the state capital cities (with lat and long) */
proc sql;
   create table Cities as
   select unique statecode as state, city, lat, long
      from maps.uscity
      where capital='Y' and statecode not in ('AK' 'PR' 'HI');
quit;
```

From this list, you can generate a links data set CitiesDist that contains the distances, in miles, between each pair of cities. The distances are calculated by using the SAS function GEODIST.

```
/* Create a list of all the possible pairs of cities */
proc sql;
   create table CitiesDist as
   select
      a.city as city1, a.lat as lat1, a.long as long1,
      b.city as city2, b.lat as lat2, b.long as long2,
      geodist(lat1, long1, lat2, long2, 'DM') as distance
      from Cities as a, Cities as b
      where a.city < b.city;
quit;
```

The following PROC OPTMODEL statements find the optimal tour through each of the capital cities:

```
/* Find optimal tour by using the network solver */
proc optmodel;
   set<str,str> CAPPAIRS;
   set<str> CAPITALS = union {<i,j> in CAPPAIRS} {i,j};
   num distance{i in CAPITALS, j in CAPITALS: i < j};
   read data CitiesDist into CAPPAIRS=[city1 city2] distance;
   set<str,str> TOUR;
   num order{CAPITALS};

   solve with NETWORK /
      loglevel  = moderate
      links     = (weight=distance)
      tsp
      out       = (order=order tour=TOUR)
   ;

   put (sum{<i,j> in TOUR} distance[i,j]);
   /* Create tour-ordered pairs (rather than input-ordered pairs) */
   str CAPbyOrder{1..card(CAPITALS)};
   for {i in CAPITALS} CAPbyOrder[order[i]] = i;
   set TSPEDGES init
      setof{i in 2..card(CAPITALS)} <CAPbyOrder[i-1],CAPbyOrder[i]>
      union {<CAPbyOrder[card(CAPITALS)],CAPbyOrder[1]>};
   num distance2{<i,j> in TSPEDGES} =
      if i < j then distance[i,j] else distance[j,i];
   create data TSPTourNodes from [node] tsp_order=order;
   create data TSPTourLinks from [city1 city2]=TSPEDGES distance=distance2;
quit;
```

The progress of the procedure is shown in Output 9.7.1. The total mileage needed to optimally traverse the capital cities is 10, 627.75 miles.

Output 9.7.1 Network Solver Log: Traveling Salesman Tour through US Capital Cities

```
NOTE: There were 1176 observations read from the data set WORK.CITIESDIST.
NOTE: The number of nodes in the input graph is 49.
NOTE: The number of links in the input graph is 1176.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 10645.918753 using 0.09
      (cpu: 0.06) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       1  10645.9187534  10040.5139714    6.03%       0
             0       1       1  10645.9187534  10241.6970024    3.95%       0
             0       1       1  10645.9187534  10262.9074205    3.73%       0
             0       1       1  10645.9187534  10350.0790852    2.86%       0
             0       1       1  10645.9187534  10381.4538838    2.55%       0
             0       1       1  10645.9187534  10470.6122886    1.67%       0
             0       1       1  10645.9187534  10492.6949171    1.46%       0
             0       1       1  10645.9187534  10519.2912679    1.20%       0
             0       1       1  10645.9187534  10576.0823291    0.66%       0
             0       1       1  10645.9187534  10590.9748294    0.52%       0
             0       1       1  10645.9187534  10607.8528157    0.36%       0
             0       1       1  10645.9187534  10607.8528157    0.36%       0
NOTE: The MILP solver added 12 cuts with 3277 cut coefficients at the root.
NOTE: Objective of the best integer solution found = 10645.918753.
NOTE: Processing the traveling salesman problem used 0.10 (cpu: 0.08) seconds.
NOTE: The TSP solver is restarting using a reduced graph with 49 nodes and 107
      links.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 10645.918753 using 0.00
      (cpu: 0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       1  10645.9187534  10040.5139714    6.03%       0
             0       1       1  10645.9187534  10241.6970024    3.95%       0
             0       1       1  10645.9187534  10262.9074205    3.73%       0
             0       1       1  10645.9187534  10350.0790852    2.86%       0
             0       1       1  10645.9187534  10381.4538838    2.55%       0
             0       1       1  10645.9187534  10470.6122886    1.67%       0
             0       1       1  10645.9187534  10497.1240512    1.42%       0
             0       1       1  10645.9187534  10519.2912679    1.20%       0
             0       1       1  10645.9187534  10576.0823291    0.66%       0
             0       1       1  10645.9187534  10590.9748294    0.52%       0
             0       1       1  10645.9187534  10607.8528157    0.36%       0
             0       1       3  10645.9187534  10607.8528157    0.36%       0
NOTE: The MILP solver added 10 cuts with 73 cut coefficients at the root.
             2       1       7  10627.7543183  10623.5771454    0.04%       0
             4       0       7  10627.7543183  10627.7543183    0.00%       0
```

Output 9.7.1 *continued*

```
NOTE: Optimal.
NOTE: Objective = 10627.754318.
NOTE: Processing the traveling salesman problem used 0.01 (cpu: 0.00) seconds.
10627.754318
NOTE: The data set WORK.TSPTOURNODES has 49 observations and 2 variables.
NOTE: The data set WORK.TSPTOURLINKS has 49 observations and 3 variables.
```

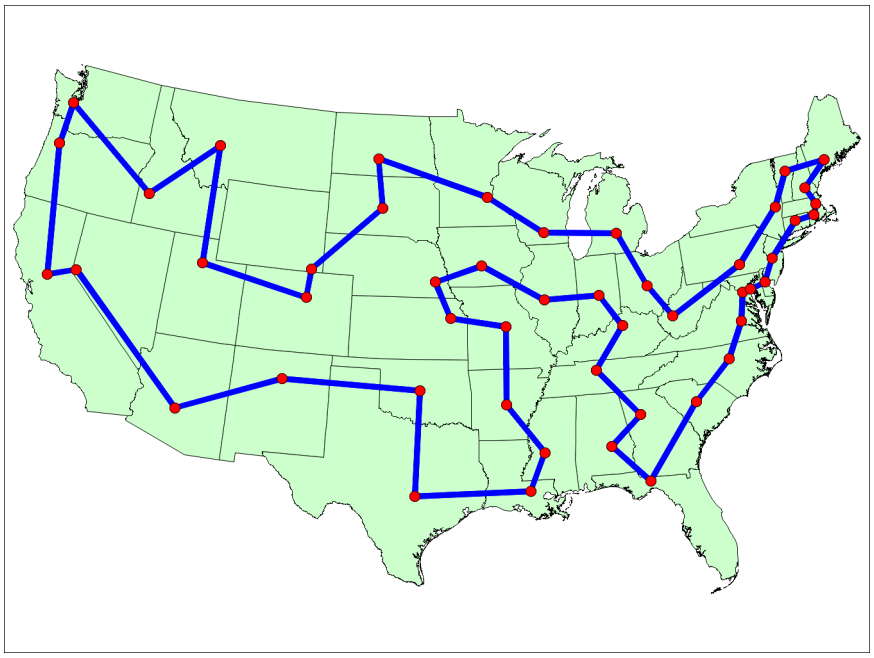
The following PROC GPROJECT and PROC GMAP statements produce a graphical display of the solution:

```
/* Merge latitude and longitude */
proc sql;
   /* merge in the lat & long for city1 */
   create table TSPTourLinksAnno1 as
   select unique TSPTourLinks.*, cities.lat as lat1, cities.long as long1
      from TSPTourLinks left join cities
      on TSPTourLinks.city1=cities.city;
   /* merge in the lat & long for city2 */
   create table TSPTourLinksAnno2 as
   select unique TSPTourLinksAnno1.*, cities.lat as lat2, cities.long as long2
      from TSPTourLinksAnno1 left join cities
      on TSPTourLinksAnno1.city2=cities.city;
quit;

/* Create the annotated data set to draw the path on the map
    (convert lat & long degrees to radians, since the map is in radians) */
data anno_path;
   set TSPTourLinksAnno2;
   length function color $8;
   xsys='2'; ysys='2'; hsys='3'; when='a'; anno_flag=1;
   function='move';
   x=atan(1)/45 * long1;
   y=atan(1)/45 * lat1;
   output;
   function='draw';
   color="blue"; size=0.8;
   x=atan(1)/45 * long2;
   y=atan(1)/45 * lat2;
   output;
run;

/* Get a map with only the contiguous 48 states */
data states;
   set maps.states (where=(fipstate(state) not in ('HI' 'AK' 'PR')));
run;

data combined;
   set states anno_path;
run;
```

```
/* Project the map and annotate the data */
proc gproject data=combined out=combined dupok;
   id state;
run;

data states anno_path;
   set combined;
   if anno_flag=1 then output anno_path;
   else                output states;
run;

/* Get a list of the endpoints locations */
proc sql;
   create table anno_dots as
   select unique x, y from anno_path;
quit;

/* Create the final annotate data set */
data anno_dots;
   set anno_dots;
   length function color $8;
   xsys='2'; ysys='2'; when='a'; hsys='3';
   function='pie';
   rotate=360; size=0.8; style='psolid'; color="red";
   output;
   style='pempty'; color="black";
   output;
run;

/* Generate the map with GMAP */
pattern1 v=s c=cxccffcc repeat=100;
proc gmap data=states map=states anno=anno_path all;
   id state;
   choro state / levels=1 nolegend coutline=black
                 anno=anno_dots des='' name="tsp";
run;
```

The minimal cost tour through the capital cities is shown on the US map in Output 9.7.2.

Output 9.7.2 Optimal Traveling Salesman Tour through US Capital Cities



The data set TSPTourLinks contains the links in the optimal tour. To display the links in the order they are to be visited, you can use the following DATA step:

```
/* Create the directed optimal tour */
data TSPTourLinksDirected(drop=next);
   set TSPTourLinks;
   retain next;
   if _N_ ne 1 and city1 ne next then do;
      city2 = city1;
      city1 = next;
   end;
   next = city2;
run;
```

The data set TSPTourLinksDirected is shown in Figure 9.84.

Figure 9.84 Links in the Optimal Traveling Salesman Tour

city1	city2	distance
Montgomery	Tallahassee	177.14
Tallahassee	Columbia	311.23
Columbia	Raleigh	182.99
Raleigh	Richmond	135.58
Richmond	Washington	97.96
Washington	Annapolis	27.89
Annapolis	Dover	54.01
Dover	Trenton	83.88
Trenton	Hartford	151.65
Hartford	Providence	65.56
Providence	Boston	38.41
Boston	Concord	66.30
Concord	Augusta	117.36
Augusta	Montpelier	139.32
Montpelier	Albany	126.19
Albany	Harrisburg	230.24
Harrisburg	Charleston	287.34
Charleston	Columbus	134.64
Columbus	Lansing	205.08
Lansing	Madison	246.88
Madison	Saint Paul	226.25
Saint Paul	Bismarck	391.25
Bismarck	Pierre	170.27
Pierre	Cheyenne	317.90
Cheyenne	Denver	98.33
Denver	Salt Lake City	373.05
Salt Lake City	Helena	403.40
Helena	Boise City	291.20
Boise City	Olympia	401.31
Olympia	Salem	146.00
Salem	Sacramento	447.40
Sacramento	Carson City	101.51
Carson City	Phoenix	577.84
Phoenix	Santa Fe	378.27
Santa Fe	Oklahoma City	474.92
Oklahoma City	Austin	357.38
Austin	Baton Rouge	394.78
Baton Rouge	Jackson	139.75
Jackson	Little Rock	206.87
Little Rock	Jefferson City	264.75
Jefferson City	Topeka	191.67
Topeka	Lincoln	132.94
Lincoln	Des Moines	168.10
Des Moines	Springfield	243.02
Springfield	Indianapolis	186.46
Indianapolis	Frankfort	129.90
Frankfort	Nashville-Davidson	175.58
Nashville-Davidson	Atlanta	212.61
Atlanta	Montgomery	145.39
		10,627.75

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice-Hall.

Applegate, D. L., Bixby, R. E., Chvátal, V., and Cook, W. J. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton, NJ: Princeton University Press.

Bron, C., and Kerbosch, J. (1973). "Algorithm 457: Finding All Cliques of an Undirected Graph." *Communications of the ACM* 16:48–50.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. Cambridge, MA, and New York: MIT Press and McGraw-Hill.

Google (2011). "Google Maps." Accessed March 16, 2011. `http://maps.google.com`.

Harley, E. R. (2003). *Graph Algorithms for Assembling Integrated Genome Maps*. Ph.D. diss., University of Toronto.

Johnson, D. B. (1975). "Finding All the Elementary Circuits of a Directed Graph." *SIAM Journal on Computing* 4:77–84.

Jonker, R., and Volgenant, A. (1987). "A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems." *Computing* 38:325–340.

Krebs, V. (2002). "Uncloaking Terrorist Networks." *First Monday* 7. http://www.firstmonday.org/issues/issue7_4/krebs/.

Kruskal, J. B. (1956). "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." *Proceedings of the American Mathematical Society* 7:48–50.

Kumar, R., and Li, H. (1994). "On Asymmetric TSP: Transformation to Symmetric TSP and Performance Bound." http://home.eng.iastate.edu/~rkumar/PUBS/atsp.pdf.

Stoer, M., and Wagner, F. (1997). "A Simple Min-Cut Algorithm." *Journal of the Association for Computing Machinery* 44:585–591.

Tarjan, R. E. (1972). "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing* 1:146–160.

Willingham, V. (2009). "Massive Transplant Effort Pairs 13 Kidneys to 13 Patients." CNN Health. Accessed March 16, 2011. http://www.cnn.com/2009/HEALTH/12/14/kidney.transplant/index.html.

Chapter 10
The Nonlinear Programming Solver

Contents

Overview: NLP Solver **487**
Getting Started: NLP Solver **489**
Syntax: NLP Solver **499**
Functional Summary 499
NLP Solver Options 500
Details: NLP Solver **506**
Basic Definitions and Notation 507
Constrained Optimization 507
Interior Point Algorithm 508
Active-Set Method 510
Multistart 512
Covariance Matrix 513
Iteration Log for the Local Solver 516
Iteration Log for Multistart 516
Solver Termination Criterion 517
Solver Termination Messages 518
Macro Variable _OROPTMODEL_ 518
Examples: NLP Solver **521**
Example 10.1: Solving Highly Nonlinear Optimization Problems 521
Example 10.2: Solving Unconstrained and Bound-Constrained Optimization Problems 523
Example 10.3: Solving NLP Problems with Range Constraints 525
Example 10.4: Solving Large-Scale NLP Problems 528
Example 10.5: Solving NLP Problems That Have Several Local Minima 530
Example 10.6: Maximum Likelihood Weibull Estimation 536
Example 10.7: Finding an Irreducible Infeasible Set 538
References **543**

Overview: NLP Solver

The sparse nonlinear programming (NLP) solver is a component of the OPTMODEL procedure that can solve optimization problems containing both nonlinear equality and inequality constraints. The general nonlinear

optimization problem can be defined as

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & h_i(x) = 0, i \in \mathcal{E} = \{1, 2, \ldots, p\} \\ & g_i(x) \geq 0, i \in \mathcal{I} = \{1, 2, \ldots, q\} \\ & l \leq x \leq u \end{array}$$

where $x \in \mathbb{R}^n$ is the vector of the decision variables; $f : \mathbb{R}^n \mapsto \mathbb{R}$ is the objective function; $h : \mathbb{R}^n \mapsto \mathbb{R}^p$ is the vector of equality constraints—that is, $h = (h_1, \ldots, h_p)$; $g : \mathbb{R}^n \mapsto \mathbb{R}^q$ is the vector of inequality constraints—that is, $g = (g_1, \ldots, g_q)$; and $l, u \in \mathbb{R}^n$ are the vectors of the lower and upper bounds, respectively, on the decision variables.

It is assumed that the functions f, h_i, and g_i are twice continuously differentiable. Any point that satisfies the constraints of the NLP problem is called a *feasible point*, and the set of all those points forms the feasible region of the NLP problem—that is, $\mathcal{F} = \{x \in \mathbb{R}^n : h(x) = 0, g(x) \geq 0, l \leq x \leq u\}$.

The NLP problem can have a unique minimum or many different minima, depending on the type of functions involved. If the objective function is convex, the equality constraint functions are linear, and the inequality constraint functions are concave, then the NLP problem is called a convex program and has a unique minimum. All other types of NLP problems are called nonconvex and can contain more than one minimum, usually called *local minima*. The solution that achieves the lowest objective value of all local minima is called the *global minimum* or *global solution* of the NLP problem. The NLP solver can find the unique minimum of convex programs and a local minimum of a general NLP problem. In addition, the solver is equipped with specific options that enable it to locate the global minimum or a good approximation of it, for those problems that contain many local minima.

The NLP solver implements the following primal-dual methods for finding a local minimum:

- interior point trust-region line-search algorithm
- active-set trust-region line-search algorithm

Both methods can solve small-, medium-, and large-scale optimization problems efficiently and robustly. These methods use exact first and second derivatives to calculate search directions. The memory requirements of both algorithms are reduced dramatically because only nonzero elements of matrices are stored. Convergence of both algorithms is achieved by using a trust-region line-search framework that guides the iterations towards the optimal solution. If a trust-region subproblem fails to provide a suitable step of improvement, a line-search is then used to fine tune the trust-region radius and ensure sufficient decrease in objective function and constraint violations.

The interior point technique implements a primal-dual interior point algorithm in which barrier functions are used to ensure that the algorithm remains feasible with respect to the bound constraints. Interior point methods are extremely useful when the optimization problem contains many inequality constraints and you suspect that most of these constraints will be satisfied as strict inequalities at the optimal solution.

The active-set technique implements an active-set algorithm in which only the inequality constraints that are satisfied as equalities, together with the original equality constraints, are considered. Once that set of constraints is identified, active-set algorithms typically converge faster than interior point algorithms. They converge faster because the size and the complexity of the original optimization problem can be reduced if only few constraints need to be considered.

For optimization problems that contain many local optima, the NLP solver can be run in multistart mode. If the multistart mode is specified, the solver samples the feasible region and generates a number of starting points. Then the local solvers can be called from each of those starting points to converge to different local optima. The local minimum with the smallest objective value is then reported back to the user as the optimal solution.

The NLP solver implements many powerful features that are obtained from recent research in the field of nonlinear optimization algorithms (Akrotirianakis and Rustem 2005; Armand, Gilbert, and Jan-Jégou 2002; Erway, Gill, and Griffin 2007; Forsgren and Gill 1998; Vanderbei 1999; Wächter and Biegler 2006; Yamashita 1998). The term *primal-dual* means that the algorithm iteratively generates better approximations of the decision variables x (usually called *primal* variables) in addition to the dual variables (also referred to as Lagrange multipliers). At every iteration, the algorithm uses a modified Newton's method to solve a system of nonlinear equations. The modifications made to Newton's method are implicitly controlled by the current trust-region radius. The solution of that system provides the direction and the steps along which the next approximation of the local minimum is searched. The active-set algorithm ensures that the primal iterations are always within their bounds—that is, $l \leq x^k \leq u$, for every iteration k. However, the interior approach relaxes this condition by using slack variables, and intermediate iterations might be infeasible.

Finally, for parameter estimation problems such as least squares, maximum likelihood, or Bayesian estimation problems, the NLP solver can calculate the covariance matrix after it successfully obtains parameter estimates.

Getting Started: NLP Solver

The NLP solver consists of two techniques that can solve a wide class of optimization problems efficiently and robustly. In this section two examples that introduce the two techniques of NLP are presented. The examples also introduce basic features of the modeling language of PROC OPTMODEL that is used to define the optimization problem.

The NLP solver can be invoked using the SOLVE statement,

SOLVE WITH NLP <*/ options*> **;**

where *options* specify the technique name, termination criteria, and how to display the results in the iteration log. For a detailed description of the *options*, see the section "NLP Solver Options" on page 500.

A Simple Problem

Consider the following simple example of a nonlinear optimization problem:

$$\begin{array}{ll} \text{minimize} & f(x) = (x_1 + 3x_2 + x_3)^2 + 4(x_1 - x_2)^2 \\ \text{subject to} & x_1 + x_2 + x_3 = 1 \\ & 6x_2 + 4x_3 - x_1^3 - 3 \geq 0 \\ & x_i \geq 0, i = 1, 2, 3 \end{array}$$

The problem consists of a quadratic objective function, a linear equality constraint, and a nonlinear inequality constraint. The goal is to find a local minimum, starting from the point $x^0 = (0.1, 0.7, 0.2)$. You can use the following call to PROC OPTMODEL to find a local minimum:

```
proc optmodel;
   var x{1..3} >= 0;
   minimize f = (x[1] + 3*x[2] + x[3])**2 + 4*(x[1] - x[2])**2;

   con constr1: sum{i in 1..3}x[i] = 1;
   con constr2: 6*x[2] + 4*x[3] - x[1]**3 - 3 >= 0;

   /* starting point */
   x[1] = 0.1;
   x[2] = 0.7;
   x[3] = 0.2;

   solve with NLP;
   print x;
quit;
```

Because no options have been specified, the default solver (INTERIORPOINT) is used to solve the problem. The SAS output displays a detailed summary of the problem along with the status of the solver at termination, the total number of iterations required, and the value of the objective function at the best feasible solution that was found. The summaries and the returned solution are shown in Figure 10.1.

Figure 10.1 Problem Summary, Solution Summary, and the Returned Solution

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Quadratic
Number of Variables	3
Bounded Above	0
Bounded Below	3
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	2
Linear LE (<=)	0
Linear EQ (=)	1
Linear GE (>=)	0
Linear Range	0
Nonlinear LE (<=)	0
Nonlinear EQ (=)	0
Nonlinear GE (>=)	1
Nonlinear Range	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Figure 10.1 *continued*

Solution Summary	
Solver	NLP
Algorithm	Interior Point
Objective Function	f
Solution Status	Best Feasible
Objective Value	1.0000158715
Optimality Error	0.1041603358
Infeasibility	2.4921244E-8
Iterations	5
Presolve Time	0.00
Solution Time	0.01

[1]	x
1	0.0000162497
2	0.0000039553
3	0.9999798200

The SAS log shown in Figure 10.2 displays a brief summary of the problem being solved, followed by the iterations that are generated by the solver.

Figure 10.2 Progress of the Algorithm as Shown in the Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 1 linear constraints (0 LE, 1 EQ, 0 GE, 0 range).
NOTE: The problem has 3 linear constraint coefficients.
NOTE: The problem has 1 nonlinear constraints (0 LE, 0 EQ, 1 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
      nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using analytic derivatives for nonlinear constraints.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
                        Objective                          Optimality
           Iter             Value     Infeasibility             Error
              0        7.20000000                 0        6.40213404
              1        1.22115550        0.00042385        0.00500000
              2        1.00188693        0.00003290        0.00480263
              3        1.00275609        0.00002123        0.00005000
              4        1.00001702   0.0000000252254        0.00187172
              5        1.00001738   0.0000000250883   0.0000005000000
NOTE: Optimal.
NOTE: Objective = 1.000017384.
NOTE: Objective of the best feasible solution found = 1.0000158715.
NOTE: The best feasible solution found is returned.
NOTE: To return the local optimal solution found, set the SOLTYPE= option to 0.
```

A Larger Optimization Problem

Consider the following larger optimization problem:

$$
\begin{array}{ll}
\text{minimize} & f(x) = \sum_{i=1}^{1000} x_i y_i + \frac{1}{2} \sum_{j=1}^{5} z_j^2 \\
\text{subject to} & x_k + y_k + \sum_{j=1}^{5} z_j = 5, \text{ for } k = 1, 2, \ldots, 1000 \\
& \sum_{i=1}^{1000} (x_i + y_i) + \sum_{j=1}^{5} z_j \geq 6 \\
& -1 \leq x_i \leq 1, i = 1, 2, \ldots, 1000 \\
& -1 \leq y_i \leq 1, i = 1, 2, \ldots, 1000 \\
& 0 \leq z_i \leq 2, i = 1, 2, \ldots, 5
\end{array}
$$

The problem consists of a quadratic objective function, 1,000 linear equality constraints, and a linear inequality constraint. There are also 2,005 variables. The goal is to find a local minimum by using the ACTIVESET technique. This can be accomplished by issuing the following call to PROC OPTMODEL:

```
proc optmodel;
   number n = 1000;
   number b = 5;
   var x{1..n} >= -1 <= 1 init  0.99;
   var y{1..n} >= -1 <= 1 init -0.99;
   var z{1..b} >=  0 <= 2 init  0.5;
   minimize f = sum {i in 1..n} x[i] * y[i] + sum {j in 1..b} 0.5 * z[j]^2;
   con cons1{k in 1..n}: x[k] + y[k] + sum {j in 1..b} z[j] = b;
   con cons2: sum {i in 1..n} (x[i] + y[i]) + sum {j in 1..b} z[j] >= b + 1;
   solve with NLP / algorithm=activeset logfreq=10;
quit;
```

The SAS output displays a detailed summary of the problem along with the status of the solver at termination, the total number of iterations required, and the value of the objective function at the local minimum. The summaries are shown in Figure 10.3.

Figure 10.3 Problem Summary and Solution Summary

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Quadratic
Number of Variables	2005
Bounded Above	0
Bounded Below	0
Bounded Below and Above	2005
Free	0
Fixed	0
Number of Constraints	1001
Linear LE (<=)	0
Linear EQ (=)	1000
Linear GE (>=)	1
Linear Range	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	NLP
Algorithm	Active Set
Objective Function	f
Solution Status	Optimal
Objective Value	-996.4999999
Optimality Error	3.632677E-7
Infeasibility	5.7544041E-8
Iterations	5
Presolve Time	0.01
Solution Time	0.19

The SAS log shown in Figure 10.4 displays a brief summary of the problem that is being solved, followed by the iterations that are generated by the solver.

Figure 10.4 Progress of the Algorithm as Shown in the Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2005 variables (0 free, 0 fixed).
NOTE: The problem has 1001 linear constraints (0 LE, 1000 EQ, 1 GE, 0 range).
NOTE: The problem has 9005 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
      nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using 2 threads for nonlinear evaluation.
NOTE: The NLP solver is called.
NOTE: The Active Set algorithm is used.
                         Objective                           Optimality
           Iter              Value     Infeasibility              Error
              0     -979.47500000        3.50000000        0.50000000
              6     -996.49999995   0.0000000503399   0.0000000262829
NOTE: Optimal.
NOTE: Objective = -996.4999999.
```

An Optimization Problem with Many Local Minima

Consider the following optimization problem:

$$\begin{aligned} \text{minimize } f(x) \quad &= \quad e^{\sin(50x)} + \sin(60e^{y}) + \sin(70\sin(x)) + \sin(\sin(80y)) \\ &\quad\; -\sin(10(x+y)) + (x^2+y^2)/4 \\ \text{subject to} \quad &\quad\; -1 \le x \le 1 \\ &\quad\; -1 \le y \le 1 \end{aligned}$$

The objective function is highly nonlinear and contains many local minima. The NLP solver provides you with the option of searching the feasible region and identifying local minima of better quality. This is achieved by writing the following SAS program:

```
proc optmodel;
  var x >= -1 <= 1;
  var y >= -1 <= 1;
  min f = exp(sin(50*x)) + sin(60*exp(y)) + sin(70*sin(x)) + sin(sin(80*y))
          - sin(10*(x+y)) + (x^2+y^2)/4;
  solve with nlp / multistart=(maxstarts=30) seed=94245;
quit;
```

The MULTISTART=() option is specified, which directs the algorithm to start the local solver from many different starting points. The SAS log is shown in Figure 10.5.

Figure 10.5 Progress of the Algorithm as Shown in the Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
      nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The MULTISTART option is enabled.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Multistart algorithm is executing in single-machine mode.
NOTE: The Multistart algorithm is using up to 4 threads.
NOTE: Random number seed 94245 is used.
```

Start		Best Objective	Local Objective	Optimality Error	Infeasibility	Local Iters	Local Status
1		-1.8567821	-1.8567821	5E-7	0	3	Optimal
2		-2.9390086	-2.9390086	5E-7	0	3	Optimal
3		-3.3068686	-3.3068686	5E-7	0	3	Optimal
4		-3.3068686	-2.6561783	5E-7	0	3	Optimal
5		-3.3068686	-1.9014527	6.14439E-7	6.14439E-7	4	Optimal
6		-3.3068686	-2.2355863	5E-7	0	3	Optimal
7		-3.3068686	-1.5173191	5E-7	0	4	Optimal
8		-3.3068686	-2.076195	5E-7	0	4	Optimal
9		-3.3068686	-0.4808983	5E-7	0	4	Optimal
10		-3.3068686	-0.4033346	5E-7	0	3	Optimal
11		-3.3068686	-2.1930943	5E-7	0	5	Optimal
12		-3.3068686	-0.7857749	5E-7	0	5	Optimal
13		-3.3068686	-0.5926643	5E-7	0	4	Optimal
14		-3.3068686	-2.9525781	5E-7	0	4	Optimal
15	*	-3.3068686	-1.3289057	5E-7	0	4	Optimal
16		-3.3068686	-2.0402058	5E-7	0	4	Optimal
17		-3.3068686	-1.5650191	5E-7	0	7	Optimal
18		-3.3068686	-2.4632393	5E-7	0	3	Optimal
19		-3.3068686	-1.404132	5E-7	0	3	Optimal
20		-3.3068686	-2.4541355	5E-7	0	4	Optimal
21		-3.3068686	-2.5171432	5E-7	0	5	Optimal
22		-3.3068686	-1.3559281	8.74345E-7	8.74345E-7	3	Optimal
23		-3.3068686	-1.031811	5E-7	0	4	Optimal
24		-3.3068686	-2.387082	5E-7	0	4	Optimal
25		-3.3068686	-0.6723829	5E-7	0	4	Optimal
26		-3.3068686	-1.2265443	5E-7	0	4	Optimal
27		-3.3068686	-1.0823455	5E-7	0	8	Optimal
28		-3.3068686	-0.6391886	5E-7	0	4	Optimal
29		-3.3068686	-0.9393803	5E-7	0	4	Optimal
30		-3.3068686	-0.5519164	5E-7	0	3	Optimal

```
NOTE: The Multistart algorithm generated 320 sample points.
NOTE: 30 distinct local optima were found.
NOTE: The best objective value found by local solver = -3.306868647.
```

Figure 10.5 *continued*

```
NOTE: The solution found by local solver with objective = -3.306868647 was
      returned.
```

The SAS log presents additional information when the MULTISTART=() option is specified. The first column counts the number of restarts of the local solver. The second column records the best local optimum that has been found so far, and the third through sixth columns record the local optimum to which the solver has converged. The final column records the status of the local solver at every iteration.

The SAS output is shown in Figure 10.6.

Figure 10.6 Problem Summary and Solution Summary

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Nonlinear
Number of Variables	2
Bounded Above	0
Bounded Below	0
Bounded Below and Above	2
Free	0
Fixed	0
Number of Constraints	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	Multistart NLP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	-3.306868647
Number of Starts	30
Number of Sample Points	320
Number of Distinct Optima	30
Random Seed Used	94245
Optimality Error	5E-7
Infeasibility	0
Presolve Time	0.00
Solution Time	3.65

A Least Squares Estimation Problem for a Regression Model

The following data are used to build a regression model:

```
data samples;
   input x1 x2 y;
   datalines;
 4  8   43.71
62  5  351.29
81 62 2878.91
85 75 3591.59
65 54 2058.71
96 84 4487.87
98 29 1773.52
36 33  767.57
30 91 1637.66
 3 59  215.28
62 57 2067.42
11 48  394.11
66 21  932.84
68 24 1069.21
95 30 1770.78
34 14  368.51
86 81 3902.27
37 49 1115.67
46 80 2136.92
87 72 3537.84
;
```

Suppose you want to compute the parameters in your regression model based on the preceding data, and the model is

$$L(a,b,c) = a * x1 + b * x2 + c * x1 * x2$$

where a, b, c are the parameters that need to be found.

The following PROC OPTMODEL call specifies the least squares problem for the regression model:

```
/* Reqression model with interactive term: y = a*x1 + b*x2 + c*x1*x2 */
proc optmodel;
   set obs;
   num x1{obs}, x2{obs}, y{obs};
   num mycov{i in 1.._nvar_, j in 1..i};
   var a, b, c;
   read data samples into obs=[_n_] x1 x2 y;
   impvar Err{i in obs} = y[i] - (a*x1[i]+b*x2[i]+c*x1[i]*x2[i]);
   min f = sum{i in obs} Err[i]^2;
   solve with nlp/covest=(cov=5 covout=mycov);
   print mycov;
   print a b c;
quit;
```

The solution is displayed in Figure 10.7.

Figure 10.7 Least Squares Problem Estimation Results

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Quadratic
Number of Variables	3
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	3
Fixed	0
Number of Constraints	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	NLP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	7.1862967833
Optimality Error	2.2796722E-9
Infeasibility	0
Iterations	17
Presolve Time	0.00
Solution Time	0.01

mycov

	1	2	3
1	0.0000047825		
2	-.0000000996	0.0000032426	
3	-.0000000676	-.0000000442	0.0000000017

a	b	c
3.0113	2.0033	0.4998

Syntax: NLP Solver

The following PROC OPTMODEL statement is available for the NLP solver:

SOLVE WITH NLP < / *options* > **;**

Functional Summary

Table 10.1 summarizes the options that can be used with the SOLVE WITH NLP statement.

Table 10.1 Options for the NLP Solver

Description	Option
Covariance Matrix Options and Suboptions	
Requests that the NLP solver compute a covariance matrix	COVEST=()
Specifies an absolute singularity criterion for matrix inversion	ASINGULAR=
Specifies the type of covariance matrix	COV=
Specifies the name of the output covariance matrix	COVOUT=
Specifies the tolerance for deciding whether a matrix is singular	COVSING=
Specifies a relative singularity criterion for matrix inversion	MSINGULAR=
Specifies a number for calculating the divisor for the covariance matrix when VARDEF=DF	NDF=
Specifies a number for calculating the scale factor for the covariance matrix	NTERMS=
Specifies a scalar factor for computing the covariance matrix	SIGSQ=
Specifies the divisor for calculating the covariance matrix	VARDEF=
Miscellaneous Option	
Specifies the seed to use to generate random numbers	SEED=
Multistart Options	
Directs the local solver to start from multiple initial points	MULTISTART=()
Specifies the maximum range of values that each variable can take during the sampling process	BNDRANGE=
Specifies the tolerance for local optima to be considered distinct	DISTTOL=
Specifies the amount of printing solution progress in multistart mode	LOGLEVEL=
Specifies the time limit in multistart mode	MAXTIME=
Specifies the maximum number of starting points to be used by the multistart algorithm	MAXSTARTS=
Optimization Option	
Specifies the optimization technique	ALGORITHM=

Table 10.1 Options for the NLP Solver (continued)

Description	Option
Output Options	
Specifies the frequency of printing solution progress (local solvers)	LOGFREQ=
Specifies the allowable types of output solution	SOLTYPE=
Solver Options	
Specifies the feasibility tolerance	FEASTOL=
Specifies the type of Hessian used by the solver	HESSTYPE=
Enables or disables IIS detection with respect to linear constraints and variable bounds	IIS=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the upper limit on the objective	OBJLIMIT=
Specifies the convergence tolerance	OPTTOL=
Specifies units of CPU time or real time	TIMETYPE=

NLP Solver Options

This section describes the options that are recognized by the NLP solver. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the NLP solver is explicitly specified using a WITH clause.

Covariance Matrix Options

COVEST=(*suboptions*)

requests that the NLP solver produce a covariance matrix. When this option is applied, the following PROC OPTMODEL options are automatically set: PRESOLVER=NONE and SOLTYPE=0. For more information, see the section "Covariance Matrix" on page 513.

You can specify the following *suboptions*:

ASINGULAR=*asing*

specifies an absolute singularity criterion for measuring the singularity of the Hessian and crossproduct Jacobian and their projected forms, which might have to be inverted to compute the covariance matrix. The value of *asing* can be any number between the machine precision and the largest positive number representable in your operating environment. The default is the square root of the machine precision. For more information, see the section "Covariance Matrix" on page 513.

COV=*number* | *string*

specifies one of six formulas for computing the covariance matrix. The formula that is used depends on the type of objective (MIN or LSQ) that is specified. Table 10.2 describes the valid values for this option and their corresponding formulas, where *nterms* is the value of the NTERMS= option and MIN, LSQ, and other symbols are defined in the section "Covariance Matrix" on page 513.

Table 10.2 Values of COV= Option

number	*string*	MIN Objective	LSQ Objective
1	M	$(nterms/d)G^{-1}JJ(f)G^{-1}$	$(nterms/d)G^{-1}VG^{-1}$
2	H	$(nterms/d)G^{-1}$	$\sigma^2 G^{-1}$
3	J	$(1/d)W^{-1}$	$\sigma^2 JJ(f)^{-1}$
4	B	$(1/d)G^{-1}WG^{-1}$	$\sigma^2 G^{-1}JJ(f)G^{-1}$
5	E	$(nterms/d)JJ(f)^{-1}$	$(1/d)V^{-1}$
6	U	$(nterms/d)W^{-1}JJ(f)W^{-1}$	$(nterms/d)JJ(f)^{-1}VJJ(f)^{-1}$

For MAX type problems, the covariance matrix is converted to MIN type by using negative Hessian, Jacobian, and function values in the computation. For more information, see the section "Covariance Matrix" on page 513.

By default, COV=2.

COVOUT=*parameter*

specifies the name of the parameter that contains the output covariance matrix. Because a covariance matrix is symmetric, you should declare the covariance matrix as either a lower-triangular matrix or a square matrix with indexes starting from 1. For example:

```
num mycov{i in 1..N, j in 1..i};   /* a  lower triangular matrix */
```

or

```
num mycov{i in 1..N, j in 1..N};   /* a square matrix */
```

where N is the number of variables.

Depending on the type of output covariance matrix, the solver updates either the lower-triangular matrix or the full square matrix. If you declare the covariance matrix as neither a lower-triangular matrix nor a square matrix, or if the indexes do not start from 1, the NLP solver issues an error message. You can use the CREATE DATA statement to output the results to a SAS data set. For more information, see the section "Covariance Matrix" on page 513.

COVSING=*covsing*

specifies a threshold, *covsing* > 0, that determines whether to consider the eigenvalues of a matrix to be 0. The value of *covsing* can be any number between the machine precision and the largest positive number representable in your operating environment. The default is set internally by the algorithm. For more information, see the section "Covariance Matrix" on page 513.

MSINGULAR=*msing*

specifies a relative singularity criterion *msing* > 0 for measuring the singularity of the Hessian and crossproduct Jacobian and their projected forms. The value of *msing* can be any number between machine precision and the largest positive number representable in your operating environment. The default is 1E–12. For more information, see the section "Covariance Matrix" on page 513.

NDF=*ndf*

specifies a number to be used in calculating the divisor d, which is used in calculating the covariance matrix when VARDEF=DF. The value of *ndf* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default is the number of optimization variables in the objective function. For more information, see the section "Covariance Matrix" on page 513.

NTERMS=*nterms*

specifies a number to be used in calculating the scale factor for the covariance matrix, as shown in Table 10.2. The value of *nterms* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default is the number of nonconstant terms in the objective function. For more information, see the section "Covariance Matrix" on page 513.

SIGSQ=*sq*

specifies a real scalar factor, $sq > 0$, for computing the covariance matrix. The value of *sq* can be any number between the machine precision and the largest positive number representable in your operating environment. For more information, see the section "Covariance Matrix" on page 513.

VARDEF=DF | N

controls how the divisor d is calculated. This divisor is used in calculating the covariance matrix and approximate standard errors. The value of d also depends on the values of the NDF= and NTERMS= options, *ndf* and *nterms*, respectively, as follows:

$$d = \begin{cases} \max(1, nterms - ndf) & \text{for VARDEF=DF} \\ nterms & \text{for VARDEF=N} \end{cases}$$

By default, VARDEF=DF if the SIGSQ= option is not specified; otherwise, by default VARDEF=N. For more information, see the section "Covariance Matrix" on page 513.

Miscellaneous Option

SEED=*N*

specifies a positive integer to be used as the seed for generating random number sequences. You can use this option to replicate results from different runs.

Multistart Options

MULTISTART=(*suboptions***)**

MS=(*suboptions***)**

enables multistart mode. In this mode, the local solver solves the problem from multiple starting points, possibly finding a better local minimum as a result. This option is disabled by default. For more information about multistart mode, see the section "Multistart" on page 512.

You can specify the following *suboptions*:

BNDRANGE=*M*

defines the range from which each variable can take values during the sampling process. This option affects only the sampling process that determines starting points for the local solver. It does not affect the bounds of the original nonlinear optimization problem. More specifically, if the ith variable x_i has lower and upper bounds ℓ_i and u_i, respectively (that is, $\ell_i \leq x_i \leq u_i$), then an initial point is generated by a sampling process as follows:

For each sample point x, the ith coordinate x_i is generated so that the following bounds hold, where x_i^0 is the default starting point or a specified starting point:

$$\begin{array}{ll} l_i \le x_i \le u_i & \text{if } l_i \text{ and } u_i \text{ are both finite} \\ l_i \le x_i \le l_i + M & \text{if only } l_i \text{ is finite} \\ u_i - M \le x_i \le u_i & \text{if only } u_i \text{ is finite} \\ x_i^0 - M/2 \le x_i \le x_i^0 + M/2 & \text{otherwise} \end{array}$$

The default value is 200 in a shared-memory computing environment and 1,000 in a distributed computing environment.

DISTTOL=ϵ

defines the tolerance by which two optimal points are considered distinct. Optimal points are considered distinct if the Euclidean distance between them is at least ϵ. The default is 1.0E–6.

LOGLEVEL=*number*

PRINTLEVEL=*number*

defines the amount of information that the multistart algorithm displays in the SAS log. Table 10.3 describes the valid values of this suboption.

Table 10.3 Values for LOGLEVEL= Suboption

number	**Description**
0	Turns off all solver-related messages to SAS log
1	Displays multistart summary information when the algorithm terminates
2	Displays multistart iteration log and summary information when the algorithm terminates
3	Displays the same information as LOGLEVEL=2 and might display additional information

By default, LOGLEVEL=2.

MAXTIME=*T*

defines the maximum allowable time *T* (in seconds) for the NLP solver to locate the best local optimum in multistart mode. The value of the TIMETYPE= option determines the type of units that are used. The time that is specified by the MAXTIME= suboption is checked only once after the completion of the local solver. Because the local solver might be called many times, the maximum time that is specified for multistart is recommended to be greater than the maximum time specified for the local solver. If you do not specify this option, the multistart algorithm does not stop based on the amount of time elapsed.

MAXSTARTS=*N*

defines the maximum number of starting points to be used for local optimization. That is, there will be no more than *N* local optimization calls in the multistart algorithm. You can specify *N* to be any nonnegative integer. When $N = 0$, the algorithm uses the default value of this option. In a shared-memory computing environment, the default value is 100. In a distributed computing environment, the default value is a number proportional to the number of threads across all the grid nodes (usually more than 100).

Optimization Options

ALGORITHM=*keyword*

TECHNIQUE=*keyword*

TECH=*keyword*

SOLVER=*keyword*

specifies the optimization technique to be used to solve the problem. The following *keywords* are valid:

INTERIORPOINT

uses a primal-dual interior point method. This technique is recommended for both small- and large-scale nonlinear optimization problems. This is the preferred solver if the problem includes a large number of inactive constraints.

ACTIVESET

uses a primal-dual active-set method. This technique is recommended for both small- and large-scale nonlinear optimization problems. This is the preferred solver if the problem includes only bound constraints or if the optimal active set can be quickly determined by the solver.

CONCURRENT (experimental)

runs the INTERIORPOINT and ACTIVESET techniques in parallel, with one thread using the INTERIORPOINT technique and the other thread using the ACTIVESET technique. The solution is returned by the first method that terminates.

The default is INTERIORPOINT.

Output Options

LOGFREQ=*N*

PRINTFREQ=*N*

specifies how often the iterations are displayed in the SAS log. *N* should be an integer between zero and the largest four-byte, signed integer, which is $2^{31} - 1$. If $N \geq 1$, the solver prints only those iterations that are a multiple of *N*. If $N = 0$, no iteration is displayed in the log. The default value is 1.

SOLTYPE=0 | **1**

specifies whether the NLP solver should return only a solution that is locally optimal. If SOLTYPE=0, the solver returns a locally optimal solution, provided it locates one. If SOLTYPE=1, the solver returns the best feasible solution found, provided its objective value is better than that of the locally optimal solution found. The default is 1.

Solver Options

FEASTOL=ϵ

defines the feasible tolerance. The solver will exit if the constraint violation is less than FEASTOL and the scaled optimality conditions are less than OPTTOL. The default is ϵ=1E–6.

HESSTYPE=FULL | **PRODUCT**

specifies the type of Hessian to be used by the solver. The valid keywords for this option are FULL and PRODUCT. If HESSTYPE=FULL, the solver uses a full Hessian. If HESSTYPE=PRODUCT, the solver uses only Hessian-vector products, not the full Hessian. When the solver uses only Hessian-vector products to find a search direction, it usually uses much less memory, especially when the

problem is large and the Hessian is not sparse. On the other hand, when the full Hessian is used, the algorithm can create a better preconditioner to solve the problem in less CPU time. The default is FULL.

IIS=*number* | *string*

specifies whether the NLP solver attempts to identify a set of linear constraints and variables that form an irreducible infeasible set (IIS). Table 10.4 describes the valid values of the IIS= option.

Table 10.4 Values for IIS= Option

number	*string*	**Description**
0	OFF	Disables IIS detection.
1	ON	Enables IIS detection.

The default is OFF.

Note that when the IIS= option is enabled, all the other NLP solver options are ignored except the following:

FEASTOL=	LOGFREQ=
LOGLEVEL=	MAXITER=
MAXTIME=	TIMETYPE=

The NLP solver ignores nonlinear constraints, if any, and invokes the LP solver's algorithm to attempt to identify an IIS. If an IIS is found, information about the infeasibilities can be found in the .status suffix values of the constraints and variables. For more information about the IIS= option, see the section "Irreducible Infeasible Set" on page 274 of Chapter 7, "The Linear Programming Solver." Also see Example 10.7 for an example that demonstrates the use of the IIS= option of the NLP solver.

MAXITER=*N*

specifies that the solver take at most *N* major iterations to determine an optimum of the NLP problem. The value of *N* is an integer between zero and the largest four-byte, signed integer, which is $2^{31} - 1$. A major iteration in NLP consists of finding a descent direction and a step size along which the next approximation of the optimum resides. The default is 5,000 iterations.

MAXTIME=*t*

specifies an upper limit of *t* units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OBJLIMIT=*M*

specifies an upper limit on the magnitude of the objective value. For a minimization problem, the algorithm terminates when the objective value becomes less than –*M*; for a maximization problem, the algorithm stops when the objective value exceeds *M*. The algorithm stopping implies that either the problem is unbounded or the algorithm diverges. If optimization were allowed to continue, numerical difficulty might be encountered. The default is *M*=1E+20. The minimum acceptable value of *M* is 1E+8. If the specified value of *M* is less than 1E+8, the value is reset to the default value 1E+20.

OPTTOL=ϵ

RELOPTTOL=ϵ

defines the measure by which you can decide whether the current iterate is an acceptable approximation of a local minimum. The value of this option is a positive real number. The NLP solver determines that the current iterate is a local minimum when the norm of the scaled vector of the optimality conditions is less than ϵ and the true constraint violation is less than FEASTOL. The default is ϵ=1E–6.

TIMETYPE=*number* | *string*

specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. Table 10.6 describes the valid values of the TIMETYPE= option.

Table 10.6 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

The "Optimization Statistics" table, an output of PROC OPTMODEL if you specify PRINTLEVEL=2 in the PROC OPTMODEL statement, also includes the same time units for Presolver Time and Solver Time. The other times (such as Problem Generation Time) in the "Optimization Statistics" table are also in the same units.

The default value of the TIMETYPE= option depends on various options. When the solver is used with distributed or multithreaded processing, then by default TIMETYPE= REAL. Otherwise, by default TIMETYPE= CPU. Table 10.7 describes the detailed logic for determining the default; the first context in the table that applies determines the default value. The NTHREADS= and NODES= options are specified in the PERFORMANCE statement of the OPTMODEL procedure. For more information about the NTHREADS= and NODES= options, see the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Table 10.7 Default Value for TIMETYPE= Option

Context	**Default**
Solver is invoked in an OPTMODEL COFOR loop	REAL
NODES= value is nonzero for multistart mode	REAL
NTHREADS= value is greater than 1	REAL
NTHREADS= 1	CPU

Details: NLP Solver

This section presents a brief discussion about the algorithmic details of the NLP solver. First, the notation is defined. Next, an introduction to the fundamental ideas in constrained optimization is presented; the main point of the second section is to present the necessary and sufficient optimality conditions, which play a central role in all optimization algorithms. The section concludes with a general overview of primal-dual

interior point and active-set algorithms for nonlinear optimization. A detailed treatment of the preceding topics can be found in Nocedal and Wright (1999), Wright (1997), and Forsgren, Gill, and Wright (2002).

Basic Definitions and Notation

The gradient of a function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is the vector of all the first partial derivatives of f and is denoted by

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)^{\mathrm{T}}$$

where the superscript T denotes the transpose of a vector.

The Hessian matrix of f, denoted by $\nabla^2 f(x)$, or simply by $H(x)$, is an $n \times n$ symmetric matrix whose (i, j) element is the second partial derivative of $f(x)$ with respect to x_i and x_j. That is, $H_{i,j}(x) = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}$.

Consider the vector function, $c : \mathbb{R}^n \mapsto \mathbb{R}^{p+q}$, whose first p elements are the equality constraint functions $h_i(x), i = 1, 2, \ldots, p$, and whose last q elements are the inequality constraint functions $g_i(x), i = 1, 2, \ldots, q$. That is,

$$c(x) = (h(x) : g(x))^{\mathrm{T}} = (h_1(x), \ldots, h_p(x) : g_1(x), \ldots, g_q(x))^{\mathrm{T}}$$

The $(p + q) \times n$ matrix whose ith row is the gradient of the ith element of $c(x)$ is called the Jacobian matrix of $c(x)$ (or simply the Jacobian of the NLP problem) and is denoted by $J(x)$. You can also use $J_h(x)$ to denote the $p \times n$ Jacobian matrix of the equality constraints and use $J_g(x)$ to denote the $q \times n$ Jacobian matrix of the inequality constraints.

Constrained Optimization

A function that plays a pivotal role in establishing conditions that characterize a local minimum of an NLP problem is the Lagrangian function $\mathcal{L}(x, y, z)$, which is defined as

$$\mathcal{L}(x, y, z) = f(x) - \sum_{i \in \mathcal{E}} y_i h_i(x) - \sum_{i \in \mathcal{I}} z_i g_i(x)$$

Note that the Lagrangian function can be seen as a linear combination of the objective and constraint functions. The coefficients of the constraints, $y_i, i \in \mathcal{E}$, and $z_i, i \in \mathcal{I}$, are called the Lagrange multipliers or dual variables. At a feasible point $\hat{x}$, an inequality constraint is called active if it is satisfied as an equality—that is, $g_i(\hat{x}) = 0$. The set of active constraints at a feasible point $\hat{x}$ is then defined as the union of the index set of the equality constraints, $\mathcal{E}$, and the indices of those inequality constraints that are active at $\hat{x}$; that is,

$$\mathcal{A}(\hat{x}) = \mathcal{E} \cup \{i \in \mathcal{I} : g_i(\hat{x}) = 0\}$$

An important condition that is assumed to hold in the majority of optimization algorithms is the so-called linear independence constraint qualification (LICQ). The LICQ states that at any feasible point $\hat{x}$, the gradients of all the active constraints are linearly independent. The main purpose of the LICQ is to ensure that the set of constraints is well-defined in a way that there are no redundant constraints or in a way that there are no constraints defined such that their gradients are always equal to zero.

The First-Order Necessary Optimality Conditions

If x^* is a local minimum of the NLP problem and the LICQ holds at x^*, then there are vectors of Lagrange multipliers y^* and z^*, with components $y_i^*, i \in \mathcal{E}$, and $z_i^*, i \in \mathcal{I}$, respectively, such that the following conditions are satisfied:

$$\begin{aligned} \nabla_x \mathcal{L}(x^*, y^*, z^*) &= 0 \\ h_i(x^*) &= 0, \quad i \in \mathcal{E} \\ g_i(x^*) &\geq 0, \quad i \in \mathcal{I} \\ z_i^* &\geq 0, \quad i \in \mathcal{I} \\ z_i^* g_i(x^*) &= 0, \quad i \in \mathcal{I} \end{aligned}$$

where $\nabla_x \mathcal{L}(x^*, y^*, z^*)$ is the gradient of the Lagrangian function with respect to x, defined as

$$\nabla_x \mathcal{L}(x^*, y^*, z^*) = \nabla f(x) - \sum_{i \in \mathcal{E}} y_i \nabla h_i(x) - \sum_{i \in \mathcal{I}} z_i \nabla g_i(x)$$

The preceding conditions are often called the *Karush-Kuhn-Tucker (KKT) conditions*. The last group of equations ($z_i g_i(x) = 0, i \in I$) is called the complementarity condition. Its main aim is to try to force the Lagrange multipliers, z_i^*, of the inactive inequalities (that is, those inequalities with $g_i(x^*) > 0$) to zero.

The KKT conditions describe the way the first derivatives of the objective and constraints are related at a local minimum x^*. However, they are not enough to fully characterize a local minimum. The second-order optimality conditions attempt to fulfill this aim by examining the curvature of the Hessian matrix of the Lagrangian function at a point that satisfies the KKT conditions.

The Second-Order Necessary Optimality Condition

Let x^* be a local minimum of the NLP problem, and let y^* and z^* be the corresponding Lagrange multipliers that satisfy the first-order optimality conditions. Then $d^{\mathrm{T}} \nabla_x^2 \mathcal{L}(x^*, y^*, z^*) d \geq 0$ for all nonzero vectors d that satisfy the following conditions:

1. $\nabla h_i^{\mathrm{T}}(x^*) d = 0, \forall i \in \mathcal{E}$
2. $\nabla g_i^{\mathrm{T}}(x^*) d = 0, \forall i \in \mathcal{A}(x^*) \cap \mathcal{I}$, such that $z_i^* > 0$
3. $\nabla g_i^{\mathrm{T}}(x^*) d \geq 0, \forall i \in \mathcal{A}(x^*) \cap \mathcal{I}$, such that $z_i^* = 0$

The second-order necessary optimality condition states that, at a local minimum, the curvature of the Lagrangian function along the directions that satisfy the preceding conditions must be nonnegative.

Interior Point Algorithm

Primal-dual interior point methods can be classified into two categories: feasible and infeasible. The first category requires that the starting point and all subsequent iterations of the algorithm strictly satisfy all the inequality constraints. The second category relaxes those requirements and allows the iterations to violate some or all of the inequality constraints during the course of the minimization procedure. The NLP solver implements an infeasible algorithm; this section concentrates on that type of algorithm.

To make the notation less cluttered and the fundamentals of interior point methods easier to understand, consider without loss of generality the following simpler NLP problem:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g_i(x) \geq 0, i \in \mathcal{I} = \{1, 2, \ldots, q\} \end{array}$$

Note that the equality and bound constraints have been omitted from the preceding problem. Initially, slack variables are added to the inequality constraints, giving rise to the problem

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g_i(x) - s_i = 0, i \in \mathcal{I} \\ & s \geq 0 \end{array}$$

where $s = (s_1, \ldots, s_q)^{\mathrm{T}}$ is the vector of slack variables, which are required to be nonnegative. Next, all the nonnegativity constraints on the slack variables are eliminated by being incorporated into the objective function, by means of a logarithmic function. This gives rise to the equality-constrained NLP problem

$$\begin{array}{ll} \text{minimize} & B(x, s) = f(x) - \mu \sum_{i \in \mathcal{I}} \ln(s_i) \\ \text{subject to} & g_i(x) - s_i = 0, i \in \mathcal{I} \end{array}$$

where μ is a positive parameter. The nonnegativity constraints on the slack variables are implicitly enforced by the logarithmic functions, since the logarithmic function prohibits s from taking zero or negative values.

Next, the equality constraints can be absorbed by using a quadratic penalty function to obtain the following:

$$\text{minimize} \quad \mathcal{M}(x, s) = f(x) + \frac{1}{2\mu} \|g(x) - s\|_2^2 - \mu \sum_{i \in \mathcal{I}} \ln(s_i)$$

The preceding unconstrained problem is often called the *penalty-barrier subproblem*. Depending on the size of the parameter μ, a local minimum of the barrier problem provides an approximation to the local minimum of the original NLP problem. The smaller the size of μ, the better the approximation becomes. Infeasible primal-dual interior point algorithms repeatedly solve the penalty-barrier problem for different values of μ that progressively go to zero, in order to get as close as possible to a local minimum of the original NLP problem.

An unconstrained minimizer of the penalty-barrier problem must satisfy the equations

$$\begin{array}{rcl} \nabla f(x) - J(x)^{\mathrm{T}} z & = & 0 \\ z - \mu S^{-1} e & = & 0 \end{array}$$

where $z = -(g(x) - s)/\mu$, $J(x)$ is the Jacobian matrix of the vector function $g(x)$, S is the diagonal matrix whose elements are the elements of the vector s (that is, $S = \mathrm{diag}\{s_1, \ldots, s_q\}$), and e is a vector of all ones. Multiplying the second equation by S and adding the definition of z as a third equation produces the following equivalent nonlinear system:

$$F^{\mu}(x, s, z) = \begin{pmatrix} \nabla f(x) - J(x)^{\mathrm{T}} z \\ Sz - e \\ g(x) - s + \mu z \end{pmatrix} = 0$$

Note that if $\mu = 0$, the preceding conditions represent the optimality conditions of the original optimization problem, after adding slack variables. One of the main aims of the algorithm is to gradually reduce the value of μ to zero, so that it converges to a local optimum of the original NLP problem. The rate by which

μ approaches zero affects the overall efficiency of the algorithm. Algorithms that treat z as an additional variable are considered primal-dual, while those that enforce the definition of $z = -(g(x) - s)/\mu$ at each iteration are consider purely primal approaches.

At iteration k, the infeasible primal-dual interior point algorithm approximately solves the preceding system by using Newton's method. The Newton system is

$$\begin{bmatrix} H_{\mathcal{L}}(x^k, z^k) & 0 & -J(x^k)^{\mathrm{T}} \\ 0 & Z^k & S^k \\ J(x^k) & -I & \mu I \end{bmatrix} \begin{bmatrix} \Delta x^k \\ \Delta s^k \\ \Delta z^k \end{bmatrix} = - \begin{bmatrix} \nabla_x f(x^k) - J(x^k)^{\mathrm{T}} z \\ -\mu e + S^k z^k \\ g(x^k) - s^k + \mu z^k \end{bmatrix}$$

where $H_{\mathcal{L}}$ is the Hessian matrix of the Lagrangian function $\mathcal{L}(x, z) = f(x) - z^{\mathrm{T}} g(x)$ of the original NLP problem; that is,

$$H_{\mathcal{L}}(x, z) = \nabla^2 f(x) - \sum_{i \in \mathcal{I}} z_i \nabla^2 g_i(x)$$

The solution $(\Delta x^k, \Delta s^k, \Delta z^k)$ of the Newton system provides a direction to move from the current iteration (x^k, s^k, z^k) to the next,

$$(x^{k+1}, s^{k+1}, z^{k+1}) = (x^k, s^k, z^k) + \alpha(\Delta x^k, \Delta s^k, \Delta z^k)$$

where α is the step length along the Newton direction. The step length is determined through a line-search procedure that ensures sufficient decrease of a merit function based on the augmented Lagrangian function of the barrier problem. The role of the merit function and the line-search procedure is to ensure that the objective and the infeasibility reduce sufficiently at every iteration and that the iterations approach a local minimum of the original NLP problem.

Active-Set Method

Active-set methods differ from interior point methods in that no barrier term is used to ensure that the algorithm remains interior with respect to the inequality constraints. Instead, attempts are made to learn the true active set. For simplicity, use the same initial slack formulation used by the interior point method description,

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g_i(x) - s_i = 0, i \in \mathcal{I} \\ & s \geq 0 \end{array}$$

where $s = (s_1, \ldots, s_q)^{\mathrm{T}}$ is the vector of slack variables, which are required to be nonnegative. Begin by absorbing the equality constraints as before into a penalty function, but keep the slack bound constraints explicitly:

$$\begin{array}{ll} \text{minimize} & \mathcal{M}(x, s) = f(x) + \dfrac{1}{2\mu} \|g(x) - s\|_2^2 \\ \text{subject to} & s \geq 0 \end{array}$$

where μ is a positive parameter. Given a solution pair $(x(\mu), s(\mu))$ for the preceding problem, you can define the active-set projection matrix P as follows:

$$P_{ij} = \begin{cases} 1 & \text{if } i = j \text{ and } s_i(\mu) = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Then $(x(\mu), s(\mu))$ is also a solution of the equality constraint subproblem:

$$\begin{aligned} &\text{minimize} \quad \mathcal{M}(x,s) = f(x) + \frac{1}{2\mu}\|g(x) - s\|_2^2 \\ &\text{subject to} \quad Ps = 0. \end{aligned}$$

The minimizer of the preceding subproblem must be a stationary point of the Lagrangian function

$$\mathcal{L}^{\mu}(x,s,z) = f(x) + \frac{1}{2\mu}\|g(x) - s\|_2^2 - z^T Ps$$

which gives the optimality equations

$$\begin{array}{rcccl} \nabla_x \mathcal{L}^{\mu}(x,s,z) & = & \nabla f(x) - J(x)^T y & = & 0 \\ \nabla_s \mathcal{L}^{\mu}(x,s,z) & = & y - P^T z & = & 0 \\ & = & Ps & = & 0 \end{array}$$

where $y = -(g(x) - s)/\mu$. Using the second equation, you can simplify the preceding equations to get the following optimality conditions for the bound-constrained penalty subproblem:

$$\begin{array}{rcl} \nabla f(x) - J(x)^T P^T z & = & 0 \\ P(g(x) - s) + \mu z & = & 0 \\ Ps & = & 0 \end{array}$$

Using the third equation directly, you can reduce the system further to

$$\begin{array}{rcl} \nabla f(x) - J(x)^T P^T z & = & 0 \\ Pg(x) + \mu z & = & 0 \end{array}$$

At iteration k, the primal-dual active-set algorithm approximately solves the preceding system by using Newton's method. The Newton system is

$$\begin{bmatrix} H_{\mathcal{L}}(x^k, z^k) & -J_{\mathcal{A}}^{\mathrm{T}} \\ J_{\mathcal{A}} & \mu I \end{bmatrix} \begin{bmatrix} \Delta x^k \\ \Delta z^k \end{bmatrix} = -\begin{bmatrix} \nabla_x f(x^k) - J_{\mathcal{A}}^{\mathrm{T}} z \\ Pg(x^k) + \mu z^k \end{bmatrix}$$

where $J_{\mathcal{A}} = PJ(x^k)$ and $H_{\mathcal{L}}$ denotes the Hessian of the Lagrangian function $f(x) - z^{\mathrm{T}} Pg(x)$. The solution $(\Delta x^k, \Delta z^k)$ of the Newton system provides a direction to move from the current iteration (x^k, s^k, z^k) to the next,

$$(x^{k+1}, z^{k+1}) = (x^k, z^k) + \alpha(\Delta x^k, \Delta z^k)$$

where α is the step length along the Newton direction. The corresponding slack variable update s^{k+1} is defined as the solution to the following subproblem whose solution can be computed analytically:

$$\begin{aligned} &\text{minimize} \quad \mathcal{M}(x^{k+1}, s) = f(x) + \frac{1}{2\mu}\|g(x^{k+1}) - s\|_2^2 \\ &\text{subject to} \quad s \geq 0 \end{aligned}$$

The step length α is then determined in a similar manner to the preceding interior point approach. At each iteration, the definition of the active-set projection matrix P is updated with respect to the new value of the constraint function $g(x^{k+1})$. For large-scale NLP, the computational bottleneck typically arises in seeking to solve the Newton system. Thus active-set methods can achieve substantial computational savings when the size of $J_{\mathcal{A}}$ is much smaller than $J(x)$; however, convergence can be slow if the active-set estimate changes combinatorially. Further, the active-set algorithm is often the superior algorithm when only bound constraints are present. In practice, both the interior point and active-set approach incorporate more sophisticated merit functions than those described in the preceding sections; however, their description is beyond the scope of this document. See Gill and Robinson (2010) for further reading.

Multistart

Frequently, nonlinear optimization problems contain many local minima because the objective or the constraints are nonconvex functions. The quality of different local minima is measured by the objective value achieved at those points. For example, if x_1^* and x_2^* are two distinct local minima and $f(x_1^*) \leq f(x_2^*)$, then x_1^* is said to be of better quality than x_2^*. The NLP solver provides a mechanism that can locate local minima of better quality by starting the local solver multiple times from different initial points. By doing so, the local solver can converge to different local minima. The local minimum with the lowest objective value is then reported back to the user.

The multistart feature consists of two phases. In the first phase, the entire feasible region is explored by generating sample points from a uniform distribution. The aim of this phase is to place at least one sample point in the region of attraction of every local minimum. Here the region of attraction of a local minimum is defined as the set of feasible points that, when used as starting points, enable a local solver to converge to that local minimum.

During the second phase, a subset of the sample points generated in the first phase is chosen by applying a clustering technique. The goal of the clustering technique is to group the initial sample points around the local minima and allow only a single local optimization to start from each cluster or group. The clustering technique aims to reduce computation time by sparing the work of unnecessarily starting multiple local optimizations within the region of attraction of the same local minimum.

The number of starting points is critical to the time spent by the solver to find a good local minimum. You can specify the maximum number of starting points by using the MAXSTARTS= suboption. If this option is not specified, the solver determines the minimum number of starting points that can provide reasonable evidence that a good local minimum will be found.

Many optimization problems contain variables with infinite upper or lower bounds. These variables can cause the sampling procedure to generate points that are not useful for locating different local minima. The efficiency of the sampling procedure can be increased by reducing the range of these variables by using the BNDRANGE= suboption. This option forces the sampling procedure to generate points that are in a smaller interval, thereby increasing the efficiency of the solver to converge to a local optimum.

The multistart feature is compatible with the PERFORMANCE statement in the OPTMODEL procedure. See Chapter 4, "Shared Concepts and Topics," for more information about the PERFORMANCE statement. The multistart feature currently supports only the DETERMINISTIC value for the PARALLELMODE= option in the PERFORMANCE statement. To ensure reproducible results, specify a nonzero value for the SEED= option.

Accessing the Starting Point That Leads to the Best Local Optimum

The starting point that leads to the best local optimum can be accessed by using the .msinit suffix in PROC OPTMODEL. In some cases, the knowledge of that starting point might be useful. For example, you can run the local solver again but this time providing as initial point the one that is stored in .msinit. This way the multistart explores a different part of the feasible region and might discover a local optimum of better quality than those found in previous runs. The use of the suffix .msinit is demonstrated in Example 10.5. For more information about suffixes in PROC OPTMODEL, see "Suffixes" on page 134 in Chapter 5, "The OPTMODEL Procedure."

Covariance Matrix

You must specify the COVEST=() option to compute an approximate covariance matrix for the parameter estimates under asymptotic theory for least squares, maximum likelihood, or Bayesian estimation, with or without corrections for degrees of freedom as specified in the VARDEF= option.

The standard form of this class of the problems is one of following:

- least squares (LSQ): $\min f(x) = s \sum_{i=1}^{m} f_i^2(x)$
- minimum or maximum (MIN/MAX): $\text{opt } f(x) = s \sum_{i=1}^{m} f_i(x)$

For example, two groups of six different forms of covariance matrices (and therefore approximate standard errors) can be computed corresponding to the following two situations, where TERMS is an index set.

- LSQ: The objective function consists solely of a positively scaled sum of squared terms, which means that least squares estimates are being computed:

$$\min f(x) = s \sum_{(i,j)\in \text{TERMS}} f_{ij}^2(x)$$

  where $s > 0$.

- MIN or MAX: The MIN or MAX declaration is specified, and the objective is not in least squares form. Together, these characteristics mean that maximum likelihood or Bayesian estimates are being computed:

$$\text{opt } f(x) = s \sum_{(i,j)\in \text{TERMS}} f_{ij}(x)$$

  where opt is either min or max and s is arbitrary.

In the preceding section, TERMS is used to denote an arbitrary index set. For example, if your problem is

$$\min z = 0.5 \sum_{i\in\mathcal{I}} (g_1^2[i] + g_2^2[i])$$

then TERMS = $\{(i, j) : i \in \mathcal{I} \text{ and } j \in \{1, 2\}\}$, where $\mathcal{I}$ is the index set of input data. The following rules apply when you specify your objective function:

- The terms $f_{ij}(x)$ are defined by using the IMPVAR declaration. The i and j values can be partitioned among observation and function indices as needed. Any number of indices can be used, including non-array indices to implicit variables.
- The objective consists of a scaled sum of terms (or squared terms for least squares). The scaling, shown as s in the preceding equations, consists of outer multiplication or division by constants of the unscaled sum of terms (or squared terms for least squares). The unary + or − operators can also be used for scaling.
- Least squares objectives require the scaling to be positive ($s > 0$). The individual f_{ij} values are scaled by $\sqrt{2s}$ by PROC OPTMODEL.

- Objectives that are not least squares allow arbitrary scaling. The scale value is distributed to the f_{ij} values.
- The summation of terms (or squared terms for least squares) is constructed with the binary $+$, SUM, and IF-THEN-ELSE operators (where IF-THEN-ELSE must have a first operand that does not depend on variables). The operands can be terms or a summation of terms (or squared terms for least squares).
- A squared term is specified as **term^2** or **term**2**.
- The terms can either be IMPVAR expression or constant expressions (expressions that do not depend on variables). Each IMPVAR element can be referenced at most once in the objective.
- The number of f_{ij} terms is determined by counting the nonconstant terms.

The following PROC OPTMODEL statements demonstrate these rules:

```
var x{VARS};
impvar g{OBS} = ...;
impvar h{OBS} = ...;

/* This objective is okay. */
min z1 = sum{i in OBS} (g[i] + h[i]);

/* This objective is okay. */
min z2 = 0.5*sum{i in OBS} (g[i]^2 + h[i]^2);

/* This objective is okay. It demonstrates multiple levels of scaling. */
min z3 = 3*(sum{i in OBS} (g[i]^2 + h[i]^2))/2;

/* This objective is okay. */
min z4 = (sum{i in OBS} (g[i]^2 + h[i]^2))/2;
```

Note that the following statements are not accepted:

```
/* This objective causes an error because individual scaling is not allowed. */
/* (division applies to inner term)                                           */
min z5 = sum{i in OBS} (g[i]^2 + h[i]^2)/2;

/* This objective causes an error because individual scaling is not allowed. */
min z6 = sum{i in OBS} 0.5*g[i]^2;

/* This objective causes an error because the element g[1] is repeated. */
min z7 = g[1] + sum{i in OBS} g[i];
```

The covariance matrix is always positive semidefinite. For MAX type problems, the covariance matrix is converted to MIN type by using negative Hessian, Jacobian, and function values in the computation. You can use the following options to check for a rank deficiency of the covariance matrix:

- The ASINGULAR= and MSINGULAR= options enable you to set two singularity criteria for the inversion of the matrix A that is needed to compute the covariance matrix, when A is either the Hessian or one of the crossproduct Jacobian matrices. The singularity criterion that is used for the inversion is

$$|d_{j,j}| \leq \max(\textit{asing}, \textit{msing} \times \max(|A_{1,1}|, \ldots, |A_{n,n}|))$$

where $d_{j,j}$ is the diagonal pivot of the matrix A, and *asing* and *msing* are the specified values of the ASINGULAR= and MSINGULAR= options, respectively.

- If the matrix A is found to be singular, the NLP solver computes a generalized inverse that satisfies Moore-Penrose conditions. The generalized inverse is computed using the computationally expensive but numerically stable eigenvalue decomposition, $A = Z\Lambda Z^T$, where Z is the orthogonal matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues, $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$. The generalized inverse of A is set to

  $$A^- = Z\Lambda^- Z^T$$

  where the diagonal matrix $\Lambda^- = \mathrm{diag}(\lambda_1^-, \ldots, \lambda_n^-)$ is defined as follows, where *covsing* is the specified value of the COVSING= option:

  $$\lambda_i^- = \begin{cases} 1/\lambda_i & \text{if } |\lambda_i| > \textit{covsing} \\ 0 & \text{if } |\lambda_i| \leq \textit{covsing} \end{cases}$$

  If the COVSING= option is not specified, then the default is $\max(\textit{asing}, \textit{msing} \times \max(|A_{1,1}|, \ldots, |A_{n,n}|))$, where *asing* and *msing* are the specified values of the ASINGULAR= and MSINGULAR= options, respectively.

For problems of the MIN or LSQ type, the matrices that are used to compute the covariance matrix are

$$G = \nabla^2 f(x)$$

$$J(f) = (\nabla f_1, \ldots, \nabla f_m) = \left(\frac{\partial f_i}{\partial x_j}\right)$$

$$JJ(f) = J(f)^T J(f)$$

$$V = J(f)^T \mathrm{diag}(f_i^2) J(f)$$

$$W = J(f)^T (\mathrm{diag}(f_i))^{-1} J(f)$$

where f_i is defined in the standard form of the covariance matrix problem. Note that when some f_i are 0, $(\mathrm{diag}(f_i))^{-1}$ is computed as a generalized inverse.

For unconstrained minimization, the formulas of the six types of covariance matrices are given in Table 10.2. The value of d in the table depends on the VARDEF= option and the values of the NDF= and NTERMS= options, *ndf* and *nterms*, respectively, as follows:

$$d = \begin{cases} \max(1, \textit{nterms} - \textit{ndf}) & \text{for VARDEF=DF} \\ \textit{nterms} & \text{for VARDEF=N} \end{cases}$$

The value of σ^2 depends on the specification of the SIGSQ= option and on the value of d,

$$\sigma^2 = \begin{cases} \textit{sq} \times \textit{nterms}/d & \text{if SIGSQ=}\textit{sq} \text{ is specified} \\ 2f(x^*)/d & \text{if SIGSQ= is not specified} \end{cases}$$

where $f(x^*)$ is the value of the objective function at the optimal solution x^*.

Because of the analytic definition, in exact arithmetic the covariance matrix is positive semidefinite at the solution. A warning message is issued if numerical computation does not result in a positive semidefinite matrix. This can happen because round-off error is introduced or the incorrect type of covariance matrix for a specified problem is selected.

Iteration Log for the Local Solver

The iteration log for the local solver provides detailed information about progress towards a locally optimal solution. This log appears when multistart mode is disabled.

The following information is displayed in the log:

Iter	indicates the iteration number.
Objective Value	indicates the objective function value.
Infeasibility	indicates the maximum value out of all constraint violations.
Optimality Error	indicates the relative optimality error (see the section "Solver Termination Criterion" on page 517).

Iteration Log for Multistart

When the MULTISTART=() option is specified, the iteration log differs from that of the local solver. More specifically, when a value of 2 is specified for the LOGLEVEL= suboption, the following information is displayed in the log:

Start	indicates the index number of each local optimization run. The following indicator can appear beside this number to provide additional information about the run: * indicates the local optimization started from a user-supplied point.
Best Objective	indicates the value of the objective function at the best local solution found so far.
Local Objective	indicates the value of the objective function obtained at the solution returned by the local solver.
Infeasibility	indicates the infeasibility error at the solution returned by the local solver.
Optimality Error	indicates the optimality error at the solution returned by the local solver.
Local Iters	indicates the number of iterations taken by the local solver.
Local Status	indicates the solution status of the local solver. Several different values can appear in this column:

OPTIMAL	indicates that the local solver found a locally optimal solution.
BESTFEASIBLE	indicates that the local solver returned the best feasible point found. See the SOLTYPE= option for more information.
INFEASIBLE	indicates that the local solver converged to a point that might be infeasible.
LOCALINFEAS	indicates that the local solver converged to a point of minimal local infeasibility.

UNBOUNDED	indicates that the local solver determined that the problem is unbounded.
ITERLIMIT	indicates that the local solver reached the maximum number of iterations and could not find a locally optimal solution.
TIMELIMIT	indicates that the local solver reached the maximum allowable time and could not find a locally optimal solution.
ABORTED	indicates that the local solver terminated due to a user interrupt.
FUNEVALERR	indicates that the local solver encountered a function evaluation error.
NUMERICERR	indicates that the local solver encountered a numerical error other than a function evaluation error.
INTERNALERR	indicates that the local solver encountered a solver system error.
OUTMEMORY	indicates that the local solver ran out of memory.
FAILED	indicates a general failure of the local solver in the absence of any other error.

Solver Termination Criterion

Because badly scaled problems can lead to slow convergence, the NLP solver dynamically rescales both the objective and constraint functions adaptively as needed. The optimality conditions are always stated with respect to the rescaled NLP. However, because typically you are most interested in the constraint violation of the original NLP, and not the internal scaled variant, you always work with respect to the true constraint violation. Thus, the solver terminates when both of the following conditions are true:

- The norm of the optimality conditions of the scaled NLP is less than the user-defined or default tolerance (OPTTOL= option).
- The norm of the constraint violation of the original NLP is less than the user-defined feasibility tolerance (FEASTOL= option).

More specifically, if

$$F(x,s,z) = (\nabla_x f(x) - J(x)^{\mathrm{T}} z, \quad Sz, \quad g(x) - s)^{\mathrm{T}}$$

is the vector of the optimality conditions of the rescaled NLP problem, then the solver terminates when

$$\| F(x,s,z) \| \leq \mathrm{OPTTOL}(1 + \|(x,s)\|)$$

and the maximum constraint violation is less than FEASTOL.

Solver Termination Messages

Upon termination, the solver produces the following messages in the log:

Optimal
The solver has successfully found a local solution to the optimization problem.

Conditionally optimal solution found
The solver is sufficiently close to a local solution, but it has difficulty in completely satisfying the user-defined optimality tolerance. This can happen when the line search finds very small steps that result in very slight progress of the algorithm. It can also happen when the prespecified tolerance is too strict for the optimization problem at hand.

Maximum number of iterations reached
The solver could not find a local optimum in the prespecified number of iterations.

Maximum specified time reached
The solver could not find a local optimum in the prespecified maximum real time for the optimization process.

Did not converge
The solver could not satisfy the optimality conditions and failed to converge.

Problem might be unbounded
The objective function takes arbitrarily large values, and the optimality conditions fail to be satisfied. This can happen when the problem is unconstrained or when the problem is constrained and the feasible region is not bounded.

Problem might be infeasible
The solver cannot identify a point in the feasible region.

Problem is infeasible
The solver detects that the problem is infeasible.

Out of memory
The problem is so large that the solver requires more memory to solve the problem.

Problem solved by the OPTMODEL presolver
The problem was solved by the OPTMODEL presolver.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro variable called _OROPTMODEL_, which contains a character string. After each PROC OPTMODEL run, you can examine this macro variable by specifying `%put &_OROPTMODEL_`; and check the execution of the most recently invoked solver from the value of the macro variable. After the NLP solver is called, the various terms of the variable are interpreted as follows:

STATUS

indicates the solver status at termination. It can take one of the following values:

OK	The solver terminated normally.
SYNTAX_ERROR	The use of syntax is incorrect.
DATA_ERROR	The input data are inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem in reading or writing of data has occurred.
SEMANTIC_ERROR	An evaluation error, such as an invalid operand type, has occurred.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take one of the following values:

IP	The interior point algorithm produced the solution data.
AS	The active-set algorithm produced the solution data.

When running algorithms concurrently (ALGORITHM=CONCURRENT), this term indicates which algorithm was the first to terminate.

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
CONDITIONAL_OPTIMAL	The optimality of the solution cannot be proven.
BEST_FEASIBLE	The solution returned is the best feasible solution. This solution status indicates that the algorithm has converged to a local optimum but a feasible (not locally optimal) solution with a better objective value has been found and hence is returned.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem might be unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
BAD_PROBLEM_TYPE	The problem type is not supported by the solver.
ITERATION_LIMIT_REACHED	The maximum allowable number of iterations has been reached.
TIME_LIMIT_REACHED	The solver reached its execution time limit.
FAILED	The solver failed to converge, possibly due to numerical issues.

OBJECTIVE
indicates the objective value that is obtained by the solver at termination.

NUMSTARTS
indicates the number of starting points. This term appears only in multistart mode.

SAMPLE_POINTS
indicates the number of points that are evaluated in the sampling phase. This term appears only in multistart mode.

DISTINCT_OPTIMA
indicates the number of distinct local optima that the solver finds. This term appears only in multistart mode.

SEED
indicates the seed value that is used to initialize the solver. This term appears only in multistart mode.

INFEASIBILITY
indicates the level of infeasibility of the constraints at the solution.

OPTIMALITY_ERROR
indicates the norm of the optimality conditions at the solution. See the section "Solver Termination Criterion" on page 517 for details.

ITERATIONS
indicates the number of iterations required to solve the problem.

PRESOLVE_TIME
indicates the real time taken for preprocessing (seconds).

SOLUTION_TIME
indicates the real time taken by the solver to perform iterations for solving the problem (seconds).

NOTE: The time that is reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

Examples: NLP Solver

Example 10.1: Solving Highly Nonlinear Optimization Problems

This example demonstrates the use of the NLP solver to solve the following highly nonlinear optimization problem, which appears in Hock and Schittkowski (1981):

$$
\begin{array}{ll}
\text{minimize} & f(x) = 0.4(x_1/x_7)^{0.67} + 0.4(x_2/x_8)^{0.67} + 10 - x_1 - x_2 \\
\text{subject to} & 1 - 0.0588x_5x_7 - 0.1x_1 \geq 0 \\
& 1 - 0.0588x_6x_8 - 0.1x_1 - 0.1x_2 \geq 0 \\
& 1 - 4x_3/x_5 - 2/(x_3^{0.71}x_5) - 0.0588x_7/x_3^{1.3} \geq 0 \\
& 1 - 4x_4/x_6 - 2/(x_4^{0.71}x_6) - 0.0588x_8/x_4^{1.3} \geq 0 \\
& 0.1 \leq f(x) \leq 4.2 \\
& 0.1 \leq x_i \leq 10, i = 1, 2, \ldots, 8
\end{array}
$$

The initial point used is $x^0 = (6, 3, 0.4, 0.2, 6, 6, 1, 0.5)$. You can call the NLP solver within PROC OPTMODEL to solve the problem by writing the following SAS statements:

```
proc optmodel;
   var x{1..8} >= 0.1 <= 10;

   min f = 0.4*(x[1]/x[7])^0.67 + 0.4*(x[2]/x[8])^0.67 + 10 - x[1] - x[2];

   con c1: 1 - 0.0588*x[5]*x[7] - 0.1*x[1] >= 0;
   con c2: 1 - 0.0588*x[6]*x[8] - 0.1*x[1] - 0.1*x[2] >= 0;
   con c3: 1 - 4*x[3]/x[5] - 2/(x[3]^0.71*x[5]) - 0.0588*x[7]/x[3]^1.3 >= 0;
   con c4: 1 - 4*x[4]/x[6] - 2/(x[4]^0.71*x[6]) - 0.0588*x[8]/x[4]^1.3 >= 0;
   con c5: 0.1 <= f <= 4.2;

   /* starting point */
   x[1] = 6;
   x[2] = 3;
   x[3] = 0.4;
   x[4] = 0.2;
   x[5] = 6;
   x[6] = 6;
   x[7] = 1;
   x[8] = 0.5;

   solve with nlp / algorithm=activeset;
   print x;
quit;
```

The summaries and the solution are shown in Output 10.1.1.

Output 10.1.1 Summaries and the Returned Solution

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Nonlinear
Number of Variables	8
Bounded Above	0
Bounded Below	0
Bounded Below and Above	8
Free	0
Fixed	0
Number of Constraints	5
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	0
Linear Range	0
Nonlinear LE (<=)	0
Nonlinear EQ (=)	0
Nonlinear GE (>=)	4
Nonlinear Range	1

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	NLP
Algorithm	Active Set
Objective Function	f
Solution Status	Optimal
Objective Value	3.9511594994
Optimality Error	3.7138163E-7
Infeasibility	9.8486624E-7
Iterations	23
Presolve Time	0.00
Solution Time	0.03

Output 10.1.1 *continued*

[1]	x
1	6.46511
2	2.23272
3	0.66740
4	0.59576
5	5.93268
6	5.52723
7	1.01332
8	0.40067

Example 10.2: Solving Unconstrained and Bound-Constrained Optimization Problems

Although the NLP techniques are suited for solving generally constrained nonlinear optimization problems, these techniques can also be used to solve unconstrained and bound-constrained problems efficiently. This example considers the relatively large nonlinear optimization problems

$$\text{minimize } f(x) = \sum_{i=1}^{n-1}(-4x_i + 3.0) + \sum_{i=1}^{n-1}(x_i^2 + x_n^2)^2$$

and

$$\begin{array}{ll} \text{minimize} & f(x) = \sum_{i=1}^{n-1} \cos(-.5x_{i+1} - x_i^2) \\ \text{subject to} & 1 \le x_i \le 2,\ i = 1, \ldots, n \end{array}$$

with $n = 100,000$. These problems are unconstrained and bound-constrained, respectively.

For large-scale problems, the default memory limit might be too small, which can lead to out-of-memory status. To prevent this occurrence, it is recommended that you set a larger memory size. See the section "Memory Limit" on page 23 for more information.

To solve the first problem, you can write the following statements:

```
proc optmodel;
   number N=100000;
   var x{1..N} init 1.0;

   minimize f = sum {i in 1..N - 1} (-4 * x[i] + 3.0)  +
                sum {i in 1..N - 1} (x[i]^2 + x[N]^2)^2;

   solve with nlp;
quit;
```

The problem and solution summaries are shown in Output 10.2.1.

Output 10.2.1 Problem Summary and Solution Summary

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Nonlinear
Number of Variables	100000
Bounded Above	0
Bounded Below	0
Bounded Below and Above	0
Free	100000
Fixed	0
Number of Constraints	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	NLP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	0
Optimality Error	1.006502E-14
Infeasibility	0
Iterations	16
Presolve Time	0.02
Solution Time	6.60

To solve the second problem, you can write the following statements (here the active-set method is specifically selected):

```
proc optmodel;
   number N=100000;
   var x{1..N} >= 1 <= 2;

   minimize f = sum {i in 1..N - 1} cos(-0.5*x[i+1] - x[i]^2);

   solve with nlp / algorithm=activeset;
quit;
```

The problem and solution summaries are shown in Output 10.2.2.

Output 10.2.2 Problem Summary and Solution Summary

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Nonlinear
Number of Variables	100000
Bounded Above	0
Bounded Below	0
Bounded Below and Above	100000
Free	0
Fixed	0
Number of Constraints	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	NLP
Algorithm	Active Set
Objective Function	f
Solution Status	Optimal
Objective Value	-99999
Optimality Error	1.0931897E-8
Infeasibility	0
Iterations	5
Presolve Time	0.02
Solution Time	6.37

Example 10.3: Solving NLP Problems with Range Constraints

Some constraints have both lower and upper bounds (that is, $a \le g(x) \le b$). These constraints are called *range constraints*. The NLP solver can handle range constraints in an efficient way. Consider the following NLP problem, taken from Hock and Schittkowski (1981),

$$
\begin{aligned}
\text{minimize} \quad & f(x) = 5.35(x_3)^2 + 0.83x_1x_5 + 37.29x_1 - 40792.141 \\
\text{subject to} \quad & 0 \le a_1 + a_2x_2x_5 + a_3x_1x_4 - a_4x_3x_5 \le 92 \\
& 0 \le a_5 + a_6x_2x_5 + a_7x_1x_2 + a_8x_3^2 - 90 \le 20 \\
& 0 \le a_9 + a_{10}x_3x_5 + a_{11}x_1x_3 + a_{12}x_3x_4 - 20 \le 5 \\
& 78 \le x_1 \le 102 \\
& 33 \le x_2 \le 45 \\
& 27 \le x_i \le 45, \quad i = 3, 4, 5
\end{aligned}
$$

where the values of the parameters $a_i, i = 1, 2, \ldots, 12$, are shown in Table 10.8.

Table 10.8 Data for Example 3

i	a_i	i	a_i	i	a_i
1	85.334407	5	80.51249	9	9.300961
2	0.0056858	6	0.0071317	10	0.0047026
3	0.0006262	7	0.0029955	11	0.0012547
4	0.0022053	8	0.0021813	12	0.0019085

The initial point used is $x^0 = (78, 33, 27, 27, 27)$. You can call the NLP solver within PROC OPTMODEL to solve this problem by writing the following statements:

```
proc optmodel;
   number l {1..5} = [78 33 27 27 27];
   number u {1..5} = [102 45 45 45 45];

   number a {1..12} =
      [85.334407 0.0056858 0.0006262 0.0022053
      80.51249 0.0071317 0.0029955 0.0021813
      9.300961 0.0047026 0.0012547 0.0019085];

   var x {j in 1..5} >= l[j] <= u[j];

   minimize f = 5.35*x[3]^2 + 0.83*x[1]*x[5] + 37.29*x[1]
                  - 40792.141;

   con constr1:
      0 <= a[1] + a[2]*x[2]*x[5] + a[3]*x[1]*x[4] -
         a[4]*x[3]*x[5] <= 92;
   con constr2:
      0 <= a[5] + a[6]*x[2]*x[5] + a[7]*x[1]*x[2] +
         a[8]*x[3]^2 - 90 <= 20;
   con constr3:
      0 <= a[9] + a[10]*x[3]*x[5] + a[11]*x[1]*x[3] +
         a[12]*x[3]*x[4] - 20 <= 5;

   x[1] = 78;
   x[2] = 33;
   x[3] = 27;
   x[4] = 27;
   x[5] = 27;

   solve with nlp / algorithm=activeset;
   print x;
quit;
```

The summaries and solution are shown in Output 10.3.1.

Output 10.3.1 Summaries and the Optimal Solution

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Quadratic
Number of Variables	5
Bounded Above	0
Bounded Below	0
Bounded Below and Above	5
Free	0
Fixed	0
Number of Constraints	3
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	0
Linear Range	0
Nonlinear LE (<=)	0
Nonlinear EQ (=)	0
Nonlinear GE (>=)	0
Nonlinear Range	3

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	NLP
Algorithm	Active Set
Objective Function	f
Solution Status	Optimal
Objective Value	-30689.1694
Optimality Error	4.1196322E-9
Infeasibility	0
Iterations	20
Presolve Time	0.00
Solution Time	0.01

[1]	x
1	78.000
2	33.000
3	29.995
4	45.000
5	36.776

Example 10.4: Solving Large-Scale NLP Problems

The following example is a selected large-scale problem from the CUTEr test set (Gould, Orban, and Toint, Ph. L. 2003) that has 20,400 variables, 20,400 lower bounds, and 9,996 linear equality constraints. This problem was selected to provide an idea of the size of problem that the NLP solver is capable of solving. In general, the maximum size of nonlinear optimization problems that can be solved with the NLP solver is controlled less by the number of variables and more by the density of the first and second derivatives of the nonlinear objective and constraint functions.

For large-scale problems, the default memory limit might be too small, which can lead to out-of-memory status. To prevent this occurrence, it is recommended that you set a larger memory size. See the section "Memory Limit" on page 23 for more information.

```
proc optmodel;
   num nx = 100;
   num ny = 100;

   var x {1..nx, 0..ny+1} >= 0;
   var y {0..nx+1, 1..ny} >= 0;

   min f = (
        sum {i in 1..nx-1, j in 1..ny-1} (x[i,j] - 1)^2
      + sum {i in 1..nx-1, j in 1..ny-1} (y[i,j] - 1)^2
      + sum {i in 1..nx-1} (x[i,ny] - 1)^2
      + sum {j in 1..ny-1} (y[nx,j] - 1)^2
      ) / 2;

   con con1 {i in 2..nx-1, j in 2..ny-1}:
      (x[i,j] - x[i-1,j]) + (y[i,j] - y[i,j-1]) = 1;
   con con2 {i in 2..nx-1}:
      x[i,0] + (x[i,1] - x[i-1,1]) + y[i,1] = 1;
   con con3 {i in 2..nx-1}:
      x[i,ny+1] + (x[i,ny] - x[i-1,ny]) - y[i,ny-1] = 1;
   con con4 {j in 2..ny-1}:
      y[0,j] + (y[1,j] - y[1,j-1]) + x[1,j] = 1;
   con con5 {j in 2..ny-1}:
      y[nx+1,j] + (y[nx,j] - y[nx,j-1]) - x[nx-1,j] = 1;

   for {i in 1..nx-1} x[i,ny].lb = 1;
   for {j in 1..ny-1} y[nx,j].lb = 1;

   solve with nlp;
quit;
```

The problem and solution summaries are shown in Output 10.4.1.

Output 10.4.1 Problem Summary and Solution Summary

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Quadratic
Number of Variables	20400
Bounded Above	0
Bounded Below	20400
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	9996
Linear LE (<=)	0
Linear EQ (=)	9996
Linear GE (>=)	0
Linear Range	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	NLP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	6237012.1224
Optimality Error	6.8574155E-7
Infeasibility	6.8574155E-7
Iterations	6
Presolve Time	0.01
Solution Time	18.15

Example 10.5: Solving NLP Problems That Have Several Local Minima

Some NLP problems contain many local minima. By default, the NLP solver converges to a single local minimum. However, the NLP solver can search the feasible region for other local minima. After it completes the search, it returns the point where the objective function achieves its minimum value. (This point might not be a local minimum; see the SOLTYPE= option for more details.) Consider the following example, taken from Hock and Schittkowski (1981):

$$
\begin{array}{ll}
\text{minimize} & f(x) = (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4 \\
\text{subject to} & x_1 + x_2^2 + x_3^3 = 2 + 3\sqrt{2} \\
& x_2 + x_4 - x_3^2 = -2 + 2\sqrt{2} \\
& x_1 x_5 = 2 \\
& -5 \le x_i \le 5, i = 1, \ldots, 5
\end{array}
$$

The following statements call the NLP solver to search the feasible region for different local minima. The PERFORMANCE statement requests that the multistart algorithm use up to four threads. The SEED= option is specified for reproducibility, but it is not required in running the multistart algorithm.

```
proc optmodel;
   var x{i in 1..5} >= -5 <= 5 init -2;

   min f=(x[1] - 1)^2 + (x[1] - x[2])^2 + (x[2] - x[3])^3 +
         (x[3] - x[4])^4 + (x[4] - x[5])^4;

   con g1: x[1] + x[2]^2 + x[3]^3 =  2 + 3*sqrt(2);
   con g2: x[2] + x[4]   - x[3]^2 = -2 + 2*sqrt(2);
   con g3: x[1]*x[5] = 2;

   performance nthreads=4;
   solve with nlp/multistart=(maxstarts=10) seed=1234;
   print x.msinit x;
quit;
```

The PRINT statement prints the starting point (x.msinit) that led to the best local solution and the best local solution (x) that the NLP solver found in multistart mode. The SAS log is shown in Output 10.5.1.

Output 10.5.1 Progress of the Algorithm as Shown in the Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 5 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 3 nonlinear constraints (0 LE, 3 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
      nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using analytic derivatives for nonlinear constraints.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The MULTISTART option is enabled.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Multistart algorithm is executing in single-machine mode.
NOTE: The Multistart algorithm is using up to 4 threads.
NOTE: Random number seed 1234 is used.
                    Best       Local  Optimality    Infeasi-  Local  Local
      Start    Objective   Objective       Error      bility  Iters  Status
          1   607.035801  607.035801  8.62585E-7  8.62585E-7      7  Optimal
          2   607.035512  607.035512  2.81991E-7  5.88394E-9      7  Optimal
          3   52.9025715  52.9025715  1.19566E-7  9.15988E-8      7  Optimal
          4   52.9025715  52.9025732  2.29572E-7  6.49817E-8     12  Optimal
          5 * 52.9025715  607.035793  5.31528E-7  4.33395E-7      8  Optimal
          6   27.8719072  27.8719072  5.95013E-7  3.05586E-7      5  Optimal
          7    27.871906   27.871906        5E-7  1.53642E-7      6  Optimal
          8    27.871906  64.8739937  8.50522E-7  2.46014E-7      7  Optimal
          9    0.0293107   0.0293107  5.07172E-7  5.07172E-7      9  Optimal
         10    0.0293107  0.02931085        5E-7  1.45889E-7     11  Optimal
NOTE: The Multistart algorithm generated 800 sample points.
NOTE: 5 distinct local optima were found.
NOTE: The best objective value found by local solver = 0.0293107008.
NOTE: The solution found by local solver with objective = 0.0293107008 was
      returned.
```

The first column in the log indicates the index of the current starting point. An additional indicator (*) can appear after the index to provide more information about the optimization run that started from the current point. For more information, see the section "Iteration Log for Multistart" on page 516. The second column records the best objective that has been found so far. Columns 3 to 6 report the objective value, optimality error, infeasibility, and number of iterations that the local solver returned when it was started from the current starting point. Finally, the last column records the status of the local solver—namely, whether it was able to converge to a local optimum from the current starting point.

The summaries and solution are shown in Output 10.5.2. Note that the best local solution was found by starting the local solver from a point at x.msinit.

Output 10.5.2 Summaries and the Optimal Solution

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Nonlinear
Number of Variables	5
Bounded Above	0
Bounded Below	0
Bounded Below and Above	5
Free	0
Fixed	0
Number of Constraints	3
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	0
Linear Range	0
Nonlinear LE (<=)	0
Nonlinear EQ (=)	3
Nonlinear GE (>=)	0
Nonlinear Range	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	Multistart NLP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	0.0293107989
Number of Starts	10
Number of Sample Points	800
Number of Distinct Optima	9
Random Seed Used	1234
Optimality Error	6.2614977E-7
Infeasibility	6.2614977E-7
Presolve Time	0.00
Solution Time	3.66

Output 10.5.2 *continued*

[1]	x.MSINIT	x
1	2.87632	1.1167
2	3.26949	1.2205
3	-1.32653	1.5378
4	-0.49448	1.9727
5	1.27877	1.7911

Alternatively, the following SAS statements show how you can add the NODES= option in the PERFORMANCE statement to run this example in distributed mode.

NOTE: SAS High-Performance Optimization software must be installed before you can invoke the MULTISTART option in distributed mode.

```
proc optmodel;
   var x{i in 1..5} >= -5 <= 5 init -2;

   min f=(x[1] - 1)^2 + (x[1] - x[2])^2 + (x[2] - x[3])^3 +
         (x[3] - x[4])^4 + (x[4] - x[5])^4;

   con g1: x[1] + x[2]^2 + x[3]^3 =  2 + 3*sqrt(2);
   con g2: x[2] + x[4]   - x[3]^2 = -2 + 2*sqrt(2);
   con g3: x[1]*x[5] = 2;

   performance nodes=4 nthreads=4;
   solve with nlp/multistart=(maxstarts=10) seed=1234;
   print x;
quit;
```

The SAS log is displayed in Output 10.5.3.

Output 10.5.3 Progress of the Algorithm as Shown in the Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 5 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 3 nonlinear constraints (0 LE, 3 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
      nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using analytic derivatives for nonlinear constraints.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The MULTISTART option is enabled.
NOTE: The Multistart algorithm is executing in the distributed computing
      environment with 4 worker nodes.
NOTE: The Multistart algorithm is using up to 4 threads.
NOTE: Random number seed 1234 is used.
                    Best       Local  Optimality    Infeasi-  Local  Local
      Start    Objective   Objective       Error      bility  Iters  Status
          1   64.8739968  64.8739968  2.99586E-7  1.19421E-7      8  Optimal
          2   52.9026071  52.9026071  3.70891E-7  3.70891E-7     11  Optimal
          3   52.9026071  607.035521  4.17297E-7  4.17297E-7      8  Optimal
          4   52.9026071  607.035871  8.43836E-7  8.43836E-7      8  Optimal
          5   52.9025794  52.9025794  6.69221E-8  3.26907E-9      8  Optimal
          6   52.9025792  52.9025792  2.39856E-7  2.84012E-8      7  Optimal
          7   52.9025015  52.9025015  9.03254E-7  9.03254E-7      5  Optimal
          8   0.02931083  0.02931083  2.53217E-7  3.54155E-9     10  Optimal
          9   0.02931083  52.9025785  4.76385E-8  1.40303E-8      7  Optimal
         10   0.02931083  52.9026134  4.08103E-7  4.08103E-7      8  Optimal
NOTE: The Multistart algorithm generated 1600 sample points.
NOTE: 7 distinct local optima were found.
NOTE: The best objective value found by local solver = 0.0293108314.
NOTE: The solution found by local solver with objective = 0.0293108314 was
      returned.
```

Output 10.5.4 shows the summaries and solution. Note that the "Performance Information" table shows that four computing nodes with four threads on each node are used in distributed mode.

Output 10.5.4 Summaries and the Optimal Solution

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Nonlinear
Number of Variables	5
Bounded Above	0
Bounded Below	0
Bounded Below and Above	5
Free	0
Fixed	0
Number of Constraints	3
Linear LE (<=)	0
Linear EQ (=)	0
Linear GE (>=)	0
Linear Range	0
Nonlinear LE (<=)	0
Nonlinear EQ (=)	3
Nonlinear GE (>=)	0
Nonlinear Range	0

Performance Information	
Host Node	<< your grid host >>
Execution Mode	Distributed
Number of Compute Nodes	4
Number of Threads per Node	4

Solution Summary	
Solver	Multistart NLP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	0.0293108314
Number of Starts	10
Number of Sample Points	1600
Number of Distinct Optima	10
Random Seed Used	1234
Optimality Error	2.5321667E-7
Infeasibility	3.5415495E-9
Presolve Time	0.00
Solution Time	1.95

Output 10.5.4 *continued*

[1]	x
1	1.1167
2	1.2204
3	1.5378
4	1.9727
5	1.7911

Example 10.6: Maximum Likelihood Weibull Estimation

The following data are taken from Lawless (1982, p. 193) and represent the number of days that it took rats that were painted with a carcinogen to develop carcinoma. The last two observations are censored data from a group of 19 rats.

```
data pike;
   input days cens @@;
   datalines;
143  0  164  0  188  0  188  0
190  0  192  0  206  0  209  0
213  0  216  0  220  0  227  0
230  0  234  0  246  0  265  0
304  0  216  1  244  1
;
```

Suppose you want to compute the maximum likelihood estimates of the scale parameter σ (α in Lawless), the shape parameter c (β in Lawless), and the location parameter θ (μ in Lawless). The observed likelihood function of the three-parameter Weibull transformation (Lawless 1982, p. 191) is

$$L(\theta,\sigma,c) = \frac{c^m}{\sigma^m}\prod_{i\in D}\left(\frac{t_i-\theta}{\sigma}\right)^{c-1}\prod_{i=1}^{n}\exp\left(-\left(\frac{t_i-\theta}{\sigma}\right)^c\right)$$

where n is the number of individuals involved in the experiment, D is the set of individuals whose lifetimes are observed, $m = |D|$, and t_i is defined by the data set. Then the log-likelihood function is

$$l(\theta,\sigma,c) = m\log c - mc\log\sigma + (c-1)\sum_{i\in D}\log(t_i-\theta) - \sum_{i=1}^{n}\left(\frac{t_i-\theta}{\sigma}\right)^c$$

For $c < 1$, the logarithmic terms become infinite as $\theta \uparrow \min_{i\in D}(t_i)$. That is, $l(\theta,\sigma,c)$ is unbounded. Thus our interest is restricted to c values greater than or equal to 1. Further, for the logarithmic terms to be defined, it is required that $\sigma > 0$ and $\theta < \min_{i\in D}(t_i)$.

The following PROC OPTMODEL call specifies the maximization of the log-likelihood function for the three-parameter Weibull estimation:

```
proc optmodel;
   set OBS;
   num days {OBS};
   num cens {OBS};
   read data pike into OBS=[_N_] days cens;
   var sig   >= 1.0e-6 init 10;
   var c     >= 1.0e-6 init 10;
   var theta >= 0 <= min {i in OBS: cens[i] = 0} days[i] init 10;

   impvar fi {i in OBS} =
      (if cens[i] = 0 then
         log(c) - c * log(sig) + (c - 1) * log(days[i] - theta)
      )
    - ((days[i] - theta) / sig)^c;
   max logf = sum {i in OBS} fi[i];

   set VARS = 1.._NVAR_;
   num mycov {i in VARS, j in 1..i};
   solve with NLP / covest=(cov=2 covout=mycov);
   print sig c theta;
   print mycov;
   create data covdata from [i j]={i in VARS, j in 1..i}
      var_i=_VAR_[i].name var_j=_VAR_[j].name mycov;
quit;
```

The solution is displayed in Output 10.6.1. The solution that the NLP solver obtains closely matches the local maximum $\theta^* = 122$, $\sigma^* = 108.4$, and $c^* = 2.712$ that are given in Lawless (1982, p. 193).

Output 10.6.1 Three-Parameter Weibull Estimation Results

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Maximization
Objective Function	logf
Objective Type	Nonlinear
Number of Variables	3
Bounded Above	0
Bounded Below	2
Bounded Below and Above	1
Free	0
Fixed	0
Number of Constraints	0

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Output 10.6.1 *continued*

Solution Summary	
Solver	NLP
Algorithm	Interior Point
Objective Function	logf
Solution Status	Optimal
Objective Value	-87.32424712
Optimality Error	5E-7
Infeasibility	0
Iterations	17
Presolve Time	0.00
Solution Time	0.03

sig	c	theta
108.38	2.7115	122.03

mycov			
	1	**2**	**3**
1	1259.9694		
2	35.5375	1.3312	
3	-1056.9867	-31.6630	977.6173

Example 10.7: Finding an Irreducible Infeasible Set

This example demonstrates the use of the IIS= option to locate an irreducible infeasible set. Suppose you have the following nonlinear programming problem:

$$
\begin{array}{lrcrcrcll}
\text{minimize} & & x_1^4 & + & x_2^4 & + & x_3^4 & & \\
\text{subject to} & & x_1 & + & x_2 & & & \geq \; 10 & \text{(c1)} \\
 & & x_1 & & & + & x_3 & \leq \; 4 & \text{(c2)} \\
 & 4 \leq & & & x_2 & + & x_3 & \leq \; 5 & \text{(c3)} \\
 & & x_1^2 & & & + & x_3 & \leq \; 5 & \text{(c4)} \\
 & & & & & x_1, & x_2 & \geq \; 0 & \\
 & & & & 0 & \leq & x_3 & \leq \; 3 &
\end{array}
$$

It is easy to verify that the following three linear constraints and one variable bound form an IIS for this problem:

$$\begin{array}{rcrcrcll} x_1 & + & x_2 & & & \geq & 10 & \text{(c1)} \\ x_1 & & & + & x_3 & \leq & 4 & \text{(c2)} \\ & & x_2 & + & x_3 & \leq & 5 & \text{(c3)} \\ & & & & x_3 & \geq & 0 & \end{array}$$

You can formulate the problem and call the NLP solver by using the following statements:

```
proc optmodel;
   /* declare variables */
   var x{1..3} >= 0;

   /* upper bound on variable x[3] */
   x[3].ub = 3;

   /* objective function */
   min f = x[1]^4 + x[2]^4 + x[3]^4;

   /* constraints */
   con c1: x[1] + x[2] >= 10;
   con c2: x[1] + x[3] <= 4;
   con c3: 4 <= x[2] + x[3] <= 5;
   con c4: x[1]^2 + x[3] <= 5;

   solve with nlp / iis = on;

   print x.status;
   print c1.status c2.status c3.status;
quit;
```

The SAS log output is shown in Output 10.7.1. Note that the PROC OPTMODEL presolver is disabled because the IIS= option is enabled. Also, a warning message is displayed to alert the user that the nonlinear constraints are ignored for the purpose of detecting an IIS.

Output 10.7.1 Finding an IIS: Original Problem

```
NOTE: The OPTMODEL presolver is disabled when the IIS= option is enabled.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 linear constraints (1 LE, 0 EQ, 1 GE, 1 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 1 nonlinear constraints (1 LE, 0 EQ, 0 GE, 0 range).
WARNING: The nonlinear constraints are ignored because the IIS= option is
         enabled.
NOTE: The NLP solver is called.
NOTE: The IIS= option is enabled.
                          Objective
      Phase Iteration        Value         Time
       P 1          1    1.400000E+01         0
       P 1          3    9.999420E-01         0
NOTE: Applying the IIS sensitivity filter.
NOTE: The sensitivity filter removed 1 constraints and 3 variable bounds.
NOTE: Applying the IIS deletion filter.
NOTE: Processing constraints.
      Processed     Removed      Time
              0           0         0
              1           0         0
              2           0         0
              3           0         0
NOTE: Processing variable bounds.
      Processed     Removed      Time
              0           0         0
              1           0         0
              2           0         0
              3           0         0
NOTE: The deletion filter removed 0 constraints and 0 variable bounds.
NOTE: The IIS= option found this problem to be infeasible.
NOTE: The IIS= option found an irreducible infeasible set with 1 variables and
      3 constraints.
NOTE: The IIS solve time is 0.02 seconds.
```

The “Solution Summary” table and the output of the PRINT statements appear in Output 10.7.2.

Output 10.7.2 Solution Summary and PRINT Statement Output

The OPTMODEL Procedure

Solution Summary	
Solver	NLP
Algorithm	IIS
Objective Function	f
Solution Status	Infeasible
Iterations	14
Iterations2	0
Presolve Time	0.00
Solution Time	0.02

[1]	x.STATUS
1	
2	
3	I_L

c1.STATUS	c2.STATUS	c3.STATUS
I_L	I_U	I_U

The "Solution Summary" table shows that the problem is infeasible. As you can see, the lower bound of variable x_3, the lower bound of constraint c1, and the upper bounds of constraints c2 and c3 form an IIS.

Making any of the components in the preceding IIS nonbinding removes the infeasibility from the IIS. Because there could be multiple IISs, you would want to remove the infeasibility from the preceding IIS and call the NLP solver with the IIS= option enabled again to see whether there is any other IIS. The following statements show how to modify the original PROC OPTMODEL statements to set the upper bound of constraint c3 to infinity, represented by CONSTANT('BIG'), and invoke the NLP IIS detection:

```
/* relax upper bound on constraint c3 */
c3.ub = constant('BIG');

solve with nlp / iis = on;

print x.status;
print c1.status c2.status c3.status;
```

The SAS log output for the modified problem is shown in Output 10.7.3.

Output 10.7.3 Finding an IIS: Modified Problem

```
NOTE: The OPTMODEL presolver is disabled when the IIS= option is enabled.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 linear constraints (1 LE, 0 EQ, 2 GE, 0 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 1 nonlinear constraints (1 LE, 0 EQ, 0 GE, 0 range).
WARNING: The nonlinear constraints are ignored because the IIS= option is
         enabled.
NOTE: The NLP solver is called.
NOTE: The IIS= option is enabled.
                            Objective
      Phase Iteration         Value         Time
       P 1          1    1.400000E+01          0
       P 1          3    0.000000E+00          0
NOTE: The IIS= option found this problem to be feasible.
NOTE: The IIS solve time is 0.02 seconds.
```

The "Solution Summary" table and the output of the PRINT statements appear in Output 10.7.4. As you can see, both the variable status and constraint status tables are empty. There is no other IIS, and the problem becomes feasible.

Output 10.7.4 Solution Summary and PRINT Statement Output

The OPTMODEL Procedure

Solution Summary	
Solver	NLP
Algorithm	IIS
Objective Function	f
Solution Status	Feasible
Iterations	3
Iterations2	0
Presolve Time	0.00
Solution Time	0.00

[1]	x.STATUS
1	
2	
3	

c1.STATUS	c2.STATUS	c3.STATUS

References

Akrotirianakis, I., and Rustem, B. (2005). "Globally Convergent Interior-Point Algorithm for Nonlinear Programming." *Journal of Optimization Theory and Applications* 125:497–521.

Armand, P., Gilbert, J. C., and Jan-Jégou, S. (2002). "A BFGS-IP Algorithm for Solving Strongly Convex Optimization Problems with Feasibility Enforced by an Exact Penalty Approach." *Mathematical Programming* 92:393–424.

Erway, J., Gill, P. E., and Griffin, J. D. (2007). "Iterative Solution of Augmented Systems Arising in Interior Point Methods." *SIAM Journal on Optimization* 18:666–690.

Forsgren, A., and Gill, P. E. (1998). "Primal-Dual Interior Methods for Nonconvex Nonlinear Programming." *SIAM Journal on Optimization* 8:1132–1152.

Forsgren, A., Gill, P. E., and Wright, M. H. (2002). "Interior Methods for Nonlinear Optimization." *SIAM Review* 44:525–597.

Gill, P. E., and Robinson, D. P. (2010). "A Primal-Dual Augmented Lagrangian." *Computational Optimization and Applications* 47:1–25.

Gould, N. I. M., Orban, D., and Toint, Ph. L. (2003). "CUTEr and SifDec: A Constrained and Unconstrained Testing Environment, Revised." *ACM Transactions on Mathematical Software* 29:373–394.

Hock, W., and Schittkowski, K. (1981). *Test Examples for Nonlinear Programming Codes.* Vol. 187 of Lecture Notes in Economics and Mathematical Systems. Berlin: Springer-Verlag.

Lawless, J. F. (1982). *Statistical Methods and Methods for Lifetime Data*. New York: John Wiley & Sons.

Nocedal, J., and Wright, S. J. (1999). *Numerical Optimization*. New York: Springer-Verlag.

Vanderbei, R. J. (1999). "LOQO: An Interior Point Code for Quadratic Programming." *Optimization Methods and Software* 11:451–484.

Wächter, A., and Biegler, L. T. (2006). "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming." *Mathematical Programming* 106:25–57.

Wright, S. J. (1997). *Primal-Dual Interior-Point Methods*. Philadelphia: SIAM.

Yamashita, H. (1998). "A Globally Convergent Primal-Dual Interior Point Method for Constrained Optimization." *Optimization Methods and Software* 10:443–469.

Chapter 11
The Quadratic Programming Solver

Contents

Overview: QP Solver **545**
Getting Started: QP Solver **547**
Syntax: QP Solver **551**
Functional Summary 551
QP Solver Options 551
Details: QP Solver **554**
Interior Point Algorithm: Overview 554
Parallel Processing 556
Iteration Log 556
Problem Statistics 556
Irreducible Infeasible Set 557
Macro Variable _OROPTMODEL_ 558
Examples: QP Solver **559**
Example 11.1: Linear Least Squares Problem 559
Example 11.2: Portfolio Optimization 562
Example 11.3: Portfolio Selection with Transactions 566
References **569**

Overview: QP Solver

The OPTMODEL procedure provides a framework for specifying and solving quadratic programs.

Mathematically, a quadratic programming (QP) problem can be stated as follows:

$$\begin{aligned} \min \quad & \tfrac{1}{2}\,\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

where

$\mathbf{Q}$	$\in$	$\mathbb{R}^{n \times n}$	is the quadratic (also known as Hessian) matrix
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the constraints matrix
$\mathbf{x}$	$\in$	$\mathbb{R}^{n}$	is the vector of decision variables
$\mathbf{c}$	$\in$	$\mathbb{R}^{n}$	is the vector of linear objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^{m}$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^{n}$	is the vector of lower bounds on the decision variables
$\mathbf{u}$	$\in$	$\mathbb{R}^{n}$	is the vector of upper bounds on the decision variables

The quadratic matrix $\mathbf{Q}$ is assumed to be symmetric; that is,

$$q_{ij} = q_{ji}, \quad \forall i, j = 1, \ldots, n$$

Indeed, it is easy to show that even if $\mathbf{Q} \neq \mathbf{Q}^{\mathrm{T}}$, then the simple modification

$$\tilde{\mathbf{Q}} = \frac{1}{2}(\mathbf{Q} + \mathbf{Q}^{\mathrm{T}})$$

produces an equivalent formulation $\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} \equiv \mathbf{x}^{\mathrm{T}}\tilde{\mathbf{Q}}\mathbf{x}$; hence symmetry is assumed. When you specify a quadratic matrix, it suffices to list only lower triangular coefficients.

In addition to being symmetric, $\mathbf{Q}$ is also required to be positive semidefinite for minimization type of models:

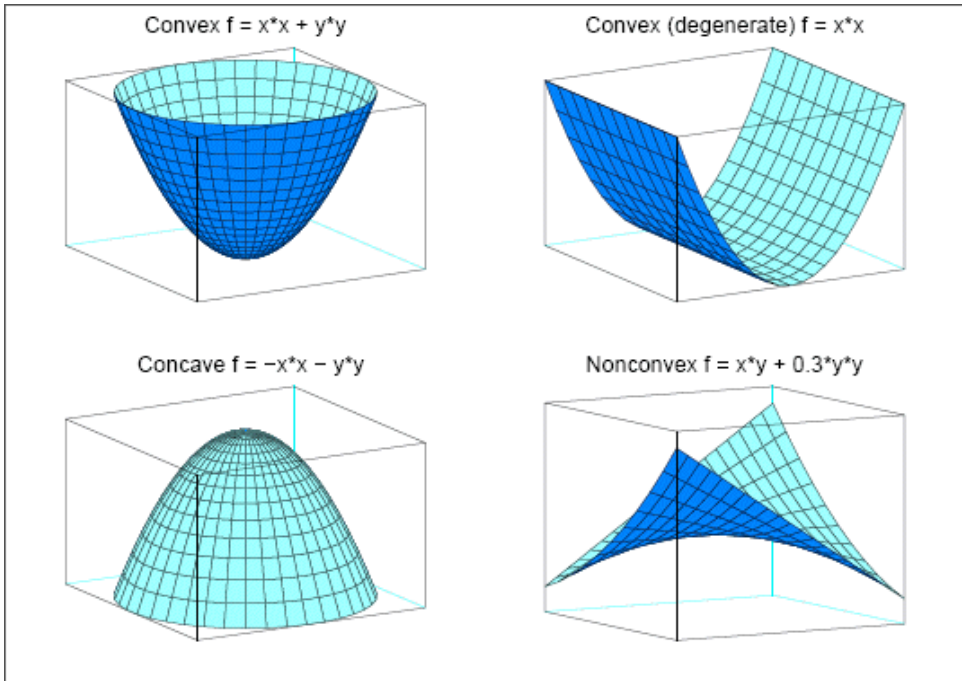
$$\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} \geq 0, \quad \forall \mathbf{x} \in \mathbb{R}^{n}$$

$\mathbf{Q}$ is required to be negative semidefinite for maximization type of models. Convexity can come as a result of a matrix-matrix multiplication

$$\mathbf{Q} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$$

or as a consequence of physical laws, and so on. See Figure 11.1 for examples of convex, concave, and nonconvex objective functions.

Figure 11.1 Examples of Convex, Concave, and Nonconvex Objective Functions



The order of constraints is insignificant. Some or all components of $\mathbf{l}$ or $\mathbf{u}$ (lower and upper bounds, respectively) can be omitted.

Getting Started: QP Solver

Consider a small illustrative example. Suppose you want to minimize a two-variable quadratic function $f(x_1, x_2)$ on the nonnegative quadrant, subject to two constraints:

$$\begin{array}{rrcrcrcrcl} \min & 2x_1 & + & 3x_2 & + & x_1^2 & + & 10x_2^2 & + & 2.5x_1x_2 \\ \text{subject to} & x_1 & - & x_2 & \leq & 1 & & & & \\ & x_1 & + & 2x_2 & \geq & 100 & & & & \\ & x_1 & & & \geq & 0 & & & & \\ & & & x_2 & \geq & 0 & & & & \end{array}$$

To use the OPTMODEL procedure, it is not necessary to fit this problem into the general QP formulation mentioned in the section "Overview: QP Solver" on page 545 and to compute the corresponding parameters. However, since these parameters are closely related to the data set that is used by the OPTQP procedure and has a quadratic programming system (QPS) format, you can compute these parameters as follows. The linear objective function coefficients, vector of right-hand sides, and lower and upper bounds are identified immediately as

$$\mathbf{c} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 100 \end{bmatrix}, \quad \mathbf{l} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} +\infty \\ +\infty \end{bmatrix}$$

Carefully construct the quadratic matrix $\mathbf{Q}$. Observe that you can use symmetry to separate the main-diagonal and off-diagonal elements:

$$\frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} \equiv \frac{1}{2}\sum_{i,j=1}^{n} x_i\, q_{ij}\, x_j = \frac{1}{2}\sum_{i=1}^{n} q_{ii}\, x_i^2 + \sum_{i>j} x_i\, q_{ij}\, x_j$$

The first expression

$$\frac{1}{2}\sum_{i=1}^{n} q_{ii}\, x_i^2$$

sums the main-diagonal elements. Thus, in this case you have

$$q_{11} = 2, \quad q_{22} = 20$$

Notice that the main-diagonal values are doubled in order to accommodate the 1/2 factor. Now the second term

$$\sum_{i>j} x_i\, q_{ij}\, x_j$$

sums the off-diagonal elements in the strict lower triangular part of the matrix. The only off-diagonal ($x_i\, x_j,\ i \neq j$) term in the objective function is $2.5\, x_1\, x_2$, so you have

$$q_{21} = 2.5$$

Notice that you do not need to specify the upper triangular part of the quadratic matrix.

Finally, the matrix of constraints is as follows:

$$\mathbf{A} = \begin{bmatrix} 1 & -1 \\ 1 & 2 \end{bmatrix}$$

The following OPTMODEL program formulates the preceding problem in a manner that is very close to the mathematical specification of the given problem:

```
/* getting started */
proc optmodel;
   var x1 >= 0; /* declare nonnegative variable x1 */
   var x2 >= 0; /* declare nonnegative variable x2 */

   /* objective: quadratic function f(x1, x2) */
   minimize f =
       /* the linear objective function coefficients */
       2 * x1 + 3 * x2 +

       /* quadratic <x, Qx> */
       x1 * x1 + 2.5 * x1 * x2 + 10 * x2 * x2;

   /* subject to the following constraints */
   con r1: x1 - x2 <= 1;
   con r2: x1 + 2 * x2 >= 100;
```

```
   /* specify iterative interior point algorithm (QP)
    * in the SOLVE statement */
   solve with qp;

   /* print the optimal solution */
   print x1 x2;
   save qps qpsdata;
quit;
```

The "with qp" clause in the SOLVE statement invokes the QP solver to solve the problem. The output is shown in Figure 11.2.

Figure 11.2 Summaries and Optimal Solution

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Quadratic
Number of Variables	2
Bounded Above	0
Bounded Below	2
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	2
Linear LE (<=)	1
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	4

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Figure 11.2 *continued*

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	15018
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Duality Gap	3.633377E-16
Complementarity	0
Iterations	6
Presolve Time	0.00
Solution Time	0.03

x1	x2
34	33

In this example, the SAVE QPS statement is used to save the QP problem in the QPS-format data set qpsdata, shown in Figure 11.3. The data set is consistent with the parameters of general quadratic programming previously computed. Also, the data set can be used as input to the OPTQP procedure.

Figure 11.3 QPS-Format Data Set

Obs	FIELD1	FIELD2	FIELD3	FIELD4	FIELD5	FIELD6
1	NAME		qpsdata	.		.
2	ROWS			.		.
3	N	f		.		.
4	L	r1		.		.
5	G	r2		.		.
6	COLUMNS			.		.
7		x1	f	2.0	r1	1
8		x1	r2	1.0		.
9		x2	f	3.0	r1	-1
10		x2	r2	2.0		.
11	RHS			.		.
12		.RHS.	r1	1.0		.
13		.RHS.	r2	100.0		.
14	QSECTION			.		.
15		x1	x1	2.0		.
16		x1	x2	2.5		.
17		x2	x2	20.0		.
18	ENDATA			.		.

Syntax: QP Solver

The following statement is available in the OPTMODEL procedure:

SOLVE WITH QP < / *options* > **;**

Functional Summary

Table 11.1 summarizes the list of options available for the SOLVE WITH QP statement, classified by function.

Table 11.1 Options for the QP Solver

Description	Option
Solver Options	
Enables or disables IIS detection	IIS=
Control Options	
Specifies the frequency of printing solution progress	LOGFREQ=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the type of presolve	PRESOLVER=
Interior Point Algorithm Options	
Specifies the stopping criterion based on duality gap	STOP_DG=
Specifies the stopping criterion based on dual infeasibility	STOP_DI=
Specifies the stopping criterion based on primal infeasibility	STOP_PI=
Specifies units of CPU time or real time	TIMETYPE=

QP Solver Options

This section describes the options recognized by the QP solver. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the QP solver is explicitly specified using a WITH clause.

The QP solver does not provide an intermediate solution if the solver terminates before reaching optimality.

Solver Options

IIS=*number* | *string*

specifies whether the QP solver attempts to identify a set of constraints and variables that form an irreducible infeasible set (IIS). Table 11.2 describes the valid values of the IIS= option.

Table 11.2 Values for IIS= Option

number	*string*	**Description**
0	OFF	Disables IIS detection.
1	ON	Enables IIS detection.

If an IIS is found, you can find information about the infeasibilities in the .status values of the constraints and variables. The default value of this option is OFF. See the section "Irreducible Infeasible Set" on page 557 for details about the IIS= option. See the section "Suffixes" on page 134 for details about the .status suffix.

Control Options

LOGFREQ=*k*

PRINTFREQ=*k*

specifies that the printing of the solution progress to the iteration log is to occur after every *k* iterations. The print frequency, *k*, is an integer between zero and the largest four-byte signed integer, which is $2^{31} - 1$.

The value $k = 0$ disables the printing of the progress of the solution. The default value of this option is 1.

MAXITER=*k*

specifies the maximum number of iterations. The value *k* can be any integer between one and the largest four-byte signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed.

MAXTIME=*t*

specifies an upper limit of *t* units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

PRESOLVER=*number* | *string*

PRESOL=*number* | *string*

specifies one of the following presolve options:

number	*string*	**Description**
0	NONE	Disables presolver.
–1	AUTOMATIC	Applies presolver by using default setting.

You can specify the PRESOLVER= value either by a character-valued option or by an integer. The default option is AUTOMATIC.

Interior Point Algorithm Options

STOP_DG=δ

specifies the desired relative duality gap, $\delta \in$ [1E–9, 1E–4]. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 554 for details.

STOP_DI=β

specifies the maximum allowed relative dual constraints violation, $\beta \in$ [1E–9, 1E–4]. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 554 for details.

STOP_PI=α

specifies the maximum allowed relative bound and primal constraints violation, $\alpha \in$ [1E–9, 1E–4]. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 554 for details.

TIMETYPE=*number* | *string*

specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. Table 11.4 describes the valid values of the TIMETYPE= option.

Table 11.4 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time.
1	REAL	Specifies units of real time.

The "Optimization Statistics" table, an output of the OPTMODEL procedure if you specify PRINTLEVEL=2 in the PROC OPTMODEL statement, also includes the same time units for Presolver Time and Solver Time. The other times (such as Problem Generation Time) in the "Optimization Statistics" table are also in the same units.

The default value of the TIMETYPE= option depends on the value of the NTHREADS= option in the PERFORMANCE statement of the OPTMODEL procedure. Table 11.5 describes the detailed logic for determining the default; the first context in the table that applies determines the default value. For more information about the NTHREADS= option, see the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Table 11.5 Default Value for TIMETYPE= Option

Context	**Default**
Solver is invoked in an OPTMODEL COFOR loop	REAL
NTHREADS= value is greater than 1	REAL
NTHREADS= 1	CPU

Details: QP Solver

Interior Point Algorithm: Overview

The QP solver implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following QP formulation (the primal):

$$\begin{aligned} \min \quad & \tfrac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

The corresponding dual formulation is

$$\begin{array}{rrcrcrcl} \max & -\frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} & + & \mathbf{b}^{\mathrm{T}}\mathbf{y} & & & & \\ \text{subject to} & -\mathbf{Q}\mathbf{x} & + & \mathbf{A}^{\mathrm{T}}\mathbf{y} & + & \mathbf{w} & = & \mathbf{c} \\ & & & & & \mathbf{y} & \geq & \mathbf{0} \\ & & & & & \mathbf{w} & \geq & \mathbf{0} \end{array}$$

where $\mathbf{y} \in \mathbb{R}^m$ refers to the vector of dual variables and $\mathbf{w} \in \mathbb{R}^n$ refers to the vector of dual slack variables.

The dual makes an important contribution to the certificate of optimality for the primal. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows where $\mathbf{e} \equiv (1, \ldots, 1)^{\mathrm{T}}$ of appropriate dimension and $\mathbf{s} \in \mathbb{R}^m$ is the vector of primal *slack* variables:

$$\begin{array}{rcll} \mathbf{A}\mathbf{x} - \mathbf{s} & = & \mathbf{b} & (\text{primal feasibility}) \\ -\mathbf{Q}\mathbf{x} + \mathbf{A}^{\mathrm{T}}\mathbf{y} + \mathbf{w} & = & \mathbf{c} & (\text{dual feasibility}) \\ \mathbf{W}\mathbf{X}\mathbf{e} & = & \mathbf{0} & (\text{complementarity}) \\ \mathbf{S}\mathbf{Y}\mathbf{e} & = & \mathbf{0} & (\text{complementarity}) \\ \mathbf{x},\, \mathbf{y},\, \mathbf{w},\, \mathbf{s} & \geq & \mathbf{0} & \end{array}$$

NOTE: Slack variables (the *s* vector) are automatically introduced by the solver when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the solver to handle slack variables much more efficiently.

The letters $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{W}$, and $\mathbf{S}$ denote matrices with corresponding x, y, w, and s on the main diagonal and zero elsewhere, as in the following example:

$$\mathbf{X} \equiv \begin{bmatrix} x_1 & 0 & \cdots & 0 \\ 0 & x_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x_n \end{bmatrix}$$

If $(\mathbf{x}^*, \mathbf{y}^*, \mathbf{w}^*, \mathbf{s}^*)$ is a solution of the previously defined system of equations that represent the KKT conditions, then $\mathbf{x}^*$ is also an optimal solution to the original QP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations,

$$\begin{bmatrix} \mathbf{Y}^{-1}\mathbf{S} & \mathbf{A} \\ \mathbf{A}^{\mathrm{T}} & -\mathbf{Q} - \mathbf{X}^{-1}\mathbf{W} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{y} \\ \Delta\mathbf{x} \end{bmatrix} = \begin{bmatrix} \Xi \\ \Theta \end{bmatrix}$$

where $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ denote the vector of *search directions* in the primal and dual spaces, respectively, and Θ and Ξ constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. The QP solver uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point algorithm is that it takes full advantage of the sparsity in the constraint and quadratic matrices, thereby enabling it to efficiently solve large-scale quadratic programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore, it is of interest to observe the following four measures where $\|v\|_2$ is the Euclidean norm of the vector v:

- relative primal infeasibility measure α:

$$\alpha = \frac{\|\mathbf{Ax} - \mathbf{b} - \mathbf{s}\|_2}{\|\mathbf{b}\|_2 + 1}$$

- relative dual infeasibility measure β:

$$\beta = \frac{\|\mathbf{Qx} + \mathbf{c} - \mathbf{A}^{\mathrm{T}}\mathbf{y} - \mathbf{w}\|_2}{\|\mathbf{c}\|_2 + 1}$$

- relative duality gap δ:

$$\delta = \frac{|\mathbf{x}^{\mathrm{T}}\mathbf{Qx} + \mathbf{c}^{\mathrm{T}}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{y}|}{|\frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Qx} + \mathbf{c}^{\mathrm{T}}\mathbf{x}| + 1}$$

- absolute complementarity γ:

$$\gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i$$

These measures are displayed in the iteration log.

Parallel Processing

The interior point algorithm can be run in single-machine mode (in single-machine mode, the computation is executed by multiple threads on a single computer). You can specify options for parallel processing in the PERFORMANCE statement, which is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Iteration Log

The following information is displayed in the iteration log:

Iter	indicates the iteration number.
Complement	indicates the (absolute) complementarity.
Duality Gap	indicates the (relative) duality gap.
Primal Infeas	indicates the (relative) primal infeasibility measure.
Bound Infeas	indicates the (relative) bound infeasibility measure.
Dual Infeas	indicates the (relative) dual infeasibility measure.
Time	indicates the time elapsed (in seconds).

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. Nonconvergence can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value that you specify in the MAXITER= or MAXTIME= option. If the complementarity or the duality gap does not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMODEL procedure causes the ODS table ProblemStatistics to be generated when the QP solver is called. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 11.4 demonstrates the contents of the ODS table ProblemStatistics.

Figure 11.4 ODS Table ProblemStatistics

The OPTMODEL Procedure

Problem Statistics	
Number of Constraint Matrix Nonzeros	4
Maximum Constraint Matrix Coefficient	2
Minimum Constraint Matrix Coefficient	1
Average Constraint Matrix Coefficient	1.25
Number of Linear Objective Nonzeros	2
Maximum Linear Objective Coefficient	3
Minimum Linear Objective Coefficient	2
Average Linear Objective Coefficient	2.5
Number of Lower Triangular Hessian Nonzeros	1
Number of Diagonal Hessian Nonzeros	2
Maximum Hessian Coefficient	20
Minimum Hessian Coefficient	2
Average Hessian Coefficient	6.75
Number of RHS Nonzeros	2
Maximum RHS	100
Minimum RHS	1
Average RHS	50.5
Maximum Number of Nonzeros per Column	2
Minimum Number of Nonzeros per Column	2
Average Number of Nonzeros per Column	2
Maximum Number of Nonzeros per Row	2
Minimum Number of Nonzeros per Row	2
Average Number of Nonzeros per Row	2

Irreducible Infeasible Set

For a quadratic programming problem, an irreducible infeasible set (IIS) is an infeasible subset of constraints and variable bounds that becomes feasible if any single constraint or variable bound is removed. It is possible to have more than one IIS in an infeasible QP. Identifying an IIS can help isolate the structural infeasibility in a QP. The IIS=ON option directs the QP solver to search for an IIS in a specified QP.

Whether a quadratic programming problem is feasible or infeasible is determined by its constraints and variable bounds, which have nothing to do with its objective function. When you specify the IIS=ON option, the QP solver treats this problem as a linear programming problem by ignoring its objective function. Then finding IIS is the same as what the LP solver does with the IIS=ON option. See the section "Irreducible Infeasible Set" on page 274 in Chapter 7, "The Linear Programming Solver," for more information about the irreducible infeasible set.

Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro called _OROPTMODEL_. This variable contains a character string. After each PROC OROPTMODEL run, you can examine this macro by specifying `%put &_OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable. The various terms of the variable after the QP solver is called are interpreted as follows.

STATUS
indicates the solver status at termination. It can take one of the following values:

OK	The solver terminated normally.
SYNTAX_ERROR	Incorrect syntax was used.
DATA_ERROR	The input data were inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem occurred in reading or writing data.
SEMANTIC_ERROR	An evaluation error, such as an invalid operand type, occurred.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM
indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take the following value:

IP	The interior point algorithm produced the solution data.

SOLUTION_STATUS
indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
CONDITIONAL_OPTIMAL	The solution is optimal, but some infeasibilities (primal, dual or bound) exceed tolerances due to scaling or pre-processing.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
BAD_PROBLEM_TYPE	The problem type is unsupported by the solver.
ITERATION_LIMIT_REACHED	The maximum allowable number of iterations was reached.
TIME_LIMIT_REACHED	The solver reached its execution time limit.
FUNCTION_CALL_LIMIT_REACHED	The solver reached its limit on function evaluations.
INTERRUPTED	The solver was interrupted externally.
FAILED	The solver failed to converge, possibly due to numerical issues.

OBJECTIVE
indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY
indicates the (relative) infeasibility of the primal constraints at the solution. See the section "Interior Point Algorithm: Overview" on page 554 for details.

DUAL_INFEASIBILITY
indicates the (relative) infeasibility of the dual constraints at the solution. See the section "Interior Point Algorithm: Overview" on page 554 for details.

BOUND_INFEASIBILITY
indicates the (relative) violation by the solution of the lower or upper bounds (or both). See the section "Interior Point Algorithm: Overview" on page 554 for details.

DUALITY_GAP
indicates the (relative) duality gap. See the section "Interior Point Algorithm: Overview" on page 554 for details.

COMPLEMENTARITY
indicates the (absolute) complementarity at the solution. See the section "Interior Point Algorithm: Overview" on page 554 for details.

ITERATIONS
indicates the number of iterations required to solve the problem.

PRESOLVE_TIME
indicates the time taken for preprocessing (in seconds).

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time that is reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

Examples: QP Solver

This section presents examples that illustrate the use of the OPTMODEL procedure to solve quadratic programming problems. Example 11.1 illustrates how to model a linear least squares problem and solve it by using PROC OPTMODEL. Example 11.2 and Example 11.3 show in detail how to model the portfolio optimization and selection problems.

Example 11.1: Linear Least Squares Problem

The linear least squares problem arises in the context of determining a solution to an overdetermined set of linear equations. In practice, these equations could arise in data fitting and estimation problems. An overdetermined system of linear equations can be defined as

$$\mathbf{Ax} = \mathbf{b}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, and $m > n$. Since this system usually does not have a solution, you need to be satisfied with some sort of approximate solution. The most widely used approximation is the least squares solution, which minimizes $\|\mathbf{Ax} - \mathbf{b}\|_2^2$.

This problem is called a least squares problem for the following reason. Let $\mathbf{A}$, $\mathbf{x}$, and $\mathbf{b}$ be defined as previously. Let $k_i(x)$ be the kth component of the vector $\mathbf{Ax} - \mathbf{b}$:

$$k_i(x) = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - b_i,\ i = 1, 2, \ldots, m$$

By definition of the Euclidean norm, the objective function can be expressed as follows:

$$\|\mathbf{Ax} - \mathbf{b}\|_2^2 = \sum_{i=1}^{m} k_i(x)^2$$

Therefore, the function you minimize is the sum of squares of m terms $k_i(x)$; hence the term least squares. The following example is an illustration of the *linear* least squares problem; that is, each of the terms k_i is a linear function of x .

Consider the following least squares problem defined by

$$\mathbf{A} = \begin{bmatrix} 4 & 0 \\ -1 & 1 \\ 3 & 2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

This translates to the following set of linear equations:

$$4x_1 = 1, \quad -x_1 + x_2 = 0, \quad 3x_1 + 2x_2 = 1$$

The corresponding least squares problem is:

$$\text{minimize} \quad (4x_1 - 1)^2 + (-x_1 + x_2)^2 + (3x_1 + 2x_2 - 1)^2$$

The preceding objective function can be expanded to:

$$\text{minimize} \quad 26x_1^2 + 5x_2^2 + 10x_1x_2 - 14x_1 - 4x_2 + 2$$

In addition, you impose the following constraint so that the equation $3x_1 + 2x_2 = 1$ is satisfied within a tolerance of 0.1:

$$0.9 \leq 3x_1 + 2x_2 \leq 1.1$$

You can use the following SAS statements to solve the least squares problem:

```
/* example 1: linear least-squares problem */
proc optmodel;
   var x1; /* declare free (no explicit bounds) variable x1 */
   var x2; /* declare free (no explicit bounds) variable x2 */
   /* declare slack variable for ranged constraint */
   var w >= 0 <= 0.2;

   /* objective function: minimize is the sum of squares */
   minimize f = 26 * x1 * x1 + 5 * x2 * x2 + 10 * x1 * x2
       - 14 * x1 - 4 * x2 + 2;

   /* subject to the following constraint */
   con L: 3 * x1 + 2 * x2 - w = 0.9;

   solve with qp;

   /* print the optimal solution */
   print x1 x2;
quit;
```

The output is shown in Output 11.1.1.

Output 11.1.1 Summaries and Optimal Solution

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Quadratic
Number of Variables	3
Bounded Above	0
Bounded Below	0
Bounded Below and Above	1
Free	2
Fixed	0
Number of Constraints	1
Linear LE (<=)	0
Linear EQ (=)	1
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	3

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Output 11.1.1 *continued*

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	0.0095238095
Primal Infeasibility	1.926295E-12
Dual Infeasibility	1.3056209E-9
Bound Infeasibility	0
Duality Gap	4.3533121E-8
Complementarity	1.3603398E-7
Iterations	3
Presolve Time	0.00
Solution Time	0.01

x1	x2
0.2381	0.1619

Example 11.2: Portfolio Optimization

Consider a portfolio optimization example. The two competing goals of investment are (1) long-term growth of capital and (2) low risk. A good portfolio grows steadily without wild fluctuations in value. The Markowitz model is an optimization model for balancing the return and risk of a portfolio. The decision variables are the amounts invested in each asset. The objective is to minimize the variance of the portfolio's total return, subject to the constraints that (1) the expected growth of the portfolio reaches at least some target level and (2) you do not invest more capital than you have.

Let $x_1, \ldots, x_n$ be the amount invested in each asset, $\mathcal{B}$ be the amount of capital you have, $\mathbf{R}$ be the random vector of asset returns over some period, and $\mathbf{r}$ be the expected value of $\mathbf{R}$. Let G be the minimum growth you hope to obtain, and $\mathcal{C}$ be the covariance matrix of $\mathbf{R}$. The objective function is $\mathrm{Var}\left(\sum_{i=1}^{n} x_i R_i\right)$, which can be equivalently denoted as $\mathbf{x}^{\mathrm{T}}\mathcal{C}\mathbf{x}$.

Assume, for example, $n = 4$. Let $\mathcal{B} = 10{,}000$, $G = 1{,}000$, $\mathbf{r} = [0.05, -0.2, 0.15, 0.30]$, and

$$\mathcal{C} = \begin{bmatrix} 0.08 & -0.05 & -0.05 & -0.05 \\ -0.05 & 0.16 & -0.02 & -0.02 \\ -0.05 & -0.02 & 0.35 & 0.06 \\ -0.05 & -0.02 & 0.06 & 0.35 \end{bmatrix}$$

The QP formulation can be written as:

$$\begin{aligned}\min \quad & 0.08x_1^2 - 0.1x_1x_2 - 0.1x_1x_3 - 0.1x_1x_4 + 0.16x_2^2 \\ & -0.04x_2x_3 - 0.04x_2x_4 + 0.35x_3^2 + 0.12x_3x_4 + 0.35x_4^2\end{aligned}$$

$$\begin{array}{lrcl}\text{subject to} & & & \\ \text{(budget)} & x_1 + x_2 + x_3 + x_4 & \leq & 10000 \\ \text{(growth)} & 0.05x_1 - 0.2x_2 + 0.15x_3 + 0.30x_4 & \geq & 1000 \\ & x_1,\ x_2,\ x_3,\ x_4 & \geq & 0\end{array}$$

Use the following SAS statements to solve the problem:

```
/* example 2: portfolio optimization */
proc optmodel;
   /* let x1, x2, x3, x4 be the amount invested in each asset */
   var x{1..4} >= 0;

   num coeff{1..4, 1..4} = [0.08 -.05 -.05 -.05
                            -.05 0.16 -.02 -.02
                            -.05 -.02 0.35 0.06
                            -.05 -.02 0.06 0.35];
   num r{1..4}=[0.05 -.20 0.15 0.30];

   /* minimize the variance of the portfolio's total return */
   minimize f = sum{i in 1..4, j in 1..4}coeff[i,j]*x[i]*x[j];

   /* subject to the following constraints */
   con BUDGET: sum{i in 1..4}x[i] <= 10000;
   con GROWTH: sum{i in 1..4}r[i]*x[i] >= 1000;

   solve with qp;

   /* print the optimal solution */
   print x;
```

The summaries and the optimal solution are shown in Output 11.2.1.

Output 11.2.1 Portfolio Optimization

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Quadratic
Number of Variables	4
Bounded Above	0
Bounded Below	4
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	2
Linear LE (<=)	1
Linear EQ (=)	0
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	8

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	2232313.4432
Primal Infeasibility	0
Dual Infeasibility	8.038873E-14
Bound Infeasibility	0
Duality Gap	4.172004E-16
Complementarity	0
Iterations	7
Presolve Time	0.00
Solution Time	0.02

[1]	x
1	3452.9
2	0.0
3	1068.8
4	2223.5

Thus, the minimum variance portfolio that earns an expected return of at least 10% is $x_1 = 3{,}452$, $x_2 = 0$, $x_3 = 1{,}068$, $x_4 = 2,223$. Asset 2 gets nothing because its expected return is -20% and its covariance with the other assets is not sufficiently negative for it to bring any diversification benefits. What if you drop the nonnegativity assumption?

Financially, that means you are allowed to short-sell—that is, sell low-mean-return assets and use the proceeds to invest in high-mean-return assets. In other words, you put a negative portfolio weight in low-mean assets and "more than 100%" in high-mean assets.

To solve the portfolio optimization problem with the short-sale option, continue to submit the following SAS statements:

```
    /* example 2: portfolio optimization with short-sale option */
    /* dropping nonnegativity assumption */
    for {i in 1..4} x[i].lb=-x[i].ub;

    solve with qp;

    /* print the optimal solution */
    print x;
quit;
```

You can see in the optimal solution displayed in Output 11.2.2 that the decision variable x_2, denoting Asset 2, is equal to $-1{,}563.61$, which means short sale of that asset.

Output 11.2.2 Portfolio Optimization with Short-Sale Option

The OPTMODEL Procedure

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	1907122.2254
Primal Infeasibility	0
Dual Infeasibility	2.755587E-13
Bound Infeasibility	0
Duality Gap	2.441695E-16
Complementarity	0
Iterations	6
Presolve Time	0.00
Solution Time	0.02

[1]	x
1	1684.35
2	-1563.61
3	682.51
4	1668.95

Example 11.3: Portfolio Selection with Transactions

Consider a portfolio selection problem with a slight modification. You are now required to take into account the current position and transaction costs associated with buying and selling assets. The objective is to find the minimum variance portfolio. In order to understand the scenario better, consider the following data.

You are given three assets. The current holding of the three assets is denoted by the vector **c** = [200, 300, 500], the amount of asset bought and sold is denoted by b_i and s_i, respectively, and the net investment in each asset is denoted by x_i and is defined by the following relation:

$$x_i - b_i + s_i = c_i, \quad i = 1, 2, 3$$

Suppose that you pay a transaction fee of 0.01 every time you buy or sell. Let the covariance matrix $\mathcal{C}$ be defined as

$$\mathcal{C} = \begin{bmatrix} 0.027489 & -0.00874 & -0.00015 \\ -0.00874 & 0.109449 & -0.00012 \\ -0.00015 & -0.00012 & 0.000766 \end{bmatrix}$$

Assume that you hope to obtain at least 12% growth. Let **r** = [1.109048, 1.169048, 1.074286] be the vector of expected return on the three assets, and let $\mathcal{B}$=1000 be the available funds. Mathematically, this problem can be written in the following manner:

$$\begin{aligned} \min \quad & 0.027489x_1^2 - 0.01748x_1x_2 - 0.0003x_1x_3 + 0.109449x_2^2 \\ & -0.00024x_2x_3 + 0.000766x_3^2 \end{aligned}$$

subject to

$$\begin{aligned} \text{(return)} \quad & \sum_{i=1}^{3} r_i x_i \geq 1.12\mathcal{B} \\ \text{(budget)} \quad & \sum_{i=1}^{3} x_i + \sum_{i=1}^{3} 0.01(b_i + s_i) = \mathcal{B} \\ \text{(balance)} \quad & x_i - b_i + s_i = c_i, \quad i = 1, 2, 3 \\ & x_i, b_i, s_i \geq 0, \quad i = 1, 2, 3 \end{aligned}$$

The problem can be solved by the following SAS statements:

```
/* example 3: portfolio selection with transactions */
proc optmodel;
   /* let x1, x2, x3 be the amount invested in each asset */
   var x{1..3} >= 0;
   /* let b1, b2, b3 be the amount of asset bought */
   var b{1..3} >= 0;
   /* let s1, s2, s3 be the amount of asset sold */
   var s{1..3} >= 0;

   /* current holdings */
   num c{1..3}=[ 200 300 500];
   /* covariance matrix */
   num coeff{1..3, 1..3} = [0.027489  -.008740  -.000150
                            -.008740  0.109449  -.000120
                            -.000150  -.000120  0.000766];
   /* returns */
   num r{1..3}=[1.109048 1.169048 1.074286];

   /* minimize the variance of the portfolio's total return */
   minimize f = sum{i in 1..3, j in 1..3}coeff[i,j]*x[i]*x[j];

   /* subject to the following constraints */
   con BUDGET: sum{i in 1..3}(x[i]+.01*b[i]+.01*s[i]) <= 1000;
   con RETURN: sum{i in 1..3}r[i]*x[i] >= 1120;
   con BALANC{i in 1..3}: x[i]-b[i]+s[i]=c[i];

   solve with qp;

   /* print the optimal solution */
   print x;
quit;
```

The output is displayed in Output 11.3.1.

Output 11.3.1 Portfolio Selection with Transactions

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Minimization
Objective Function	f
Objective Type	Quadratic
Number of Variables	9
Bounded Above	0
Bounded Below	9
Bounded Below and Above	0
Free	0
Fixed	0
Number of Constraints	5
Linear LE (<=)	1
Linear EQ (=)	3
Linear GE (>=)	1
Linear Range	0
Constraint Coefficients	21

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	f
Solution Status	Optimal
Objective Value	19560.725753
Primal Infeasibility	7.000061E-17
Dual Infeasibility	0
Bound Infeasibility	0
Duality Gap	2.994187E-14
Complementarity	0
Iterations	11
Presolve Time	0.00
Solution Time	0.02

[1]	x
1	397.58
2	406.12
3	190.17

References

Freund, R. W. (1991). "On Polynomial Preconditioning and Asymptotic Convergence Factors for Indefinite Hermitian Matrices." *Linear Algebra and Its Applications* 154–156:259–288.

Freund, R. W., and Jarre, F. (1997). "A QMR-Based Interior Point Algorithm for Solving Linear Programs." *Mathematical Programming* 76:183–210.

Freund, R. W., and Nachtigal, N. M. (1996). "QMRPACK: A Package of QMR Algorithms." *ACM Transactions on Mathematical Software* 22:46–77.

Vanderbei, R. J. (1999). "LOQO: An Interior Point Code for Quadratic Programming." *Optimization Methods and Software* 11:451–484.

Wright, S. J. (1997). *Primal-Dual Interior-Point Methods*. Philadelphia: SIAM.

Chapter 12
The OPTLP Procedure

Contents

Overview: OPTLP Procedure **572**
Getting Started: OPTLP Procedure **572**
Syntax: OPTLP Procedure **574**
Functional Summary 575
PROC OPTLP Statement 576
Decomposition Algorithm Statements 582
PERFORMANCE Statement 582
Details: OPTLP Procedure **582**
Data Input and Output 582
Presolve 586
Pricing Strategies for the Primal and Dual Simplex Algorithms 587
Warm Start for the Primal and Dual Simplex Algorithms 587
The Network Simplex Algorithm 588
The Interior Point Algorithm 588
Iteration Log for the Primal and Dual Simplex Algorithms 590
Iteration Log for the Network Simplex Algorithm 591
Iteration Log for the Interior Point Algorithm 592
Iteration Log for the Crossover Algorithm 592
Concurrent LP 593
Parallel Processing 593
ODS Tables 593
Irreducible Infeasible Set 597
Macro Variable _OROPTLP_ 598
Examples: OPTLP Procedure **600**
Example 12.1: Oil Refinery Problem 600
Example 12.2: Using the Interior Point Algorithm 604
Example 12.3: The Diet Problem 605
Example 12.4: Reoptimizing after Modifying the Objective Function 607
Example 12.5: Reoptimizing after Modifying the Right-Hand Side 610
Example 12.6: Reoptimizing after Adding a New Constraint 612
Example 12.7: Finding an Irreducible Infeasible Set 615
Example 12.8: Using the Network Simplex Algorithm 620
References **623**

Overview: OPTLP Procedure

The OPTLP procedure provides four methods of solving linear programs (LPs). A linear program has the following formulation:

$$\begin{aligned} \min \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of decision variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the matrix of constraints
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^n$	is the vector of lower bounds on variables
$\mathbf{u}$	$\in$	$\mathbb{R}^n$	is the vector of upper bounds on variables

The following LP algorithms are available in the OPTLP procedure:

- primal simplex algorithm
- dual simplex algorithm
- network simplex algorithm
- interior point algorithm

The primal and dual simplex algorithms implement the two-phase simplex method. In phase I, the algorithm tries to find a feasible solution. If no feasible solution is found, the LP is infeasible; otherwise, the algorithm enters phase II to solve the original LP. The network simplex algorithm extracts a network substructure, solves this using network simplex, and then constructs an advanced basis to feed to either primal or dual simplex. The interior point algorithm implements a primal-dual predictor-corrector interior point algorithm.

PROC OPTLP requires a linear program to be specified using a SAS data set that adheres to the MPS format, a widely accepted format in the optimization community. For details about the MPS format see Chapter 17, "The MPS-Format SAS Data Set."

You can use the MPSOUT= option to convert typical PROC LP format data sets into MPS-format SAS data sets. The option is available in the LP, INTPOINT, and NETFLOW procedures. For details about this option, see Chapter 4, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), Chapter 3, "The INTPOINT Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), and Chapter 5, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

Getting Started: OPTLP Procedure

The following example illustrates how you can use the OPTLP procedure to solve linear programs. Suppose you want to solve the following problem:

$$
\begin{array}{rrcrcrcrl}
\min & 2x_1 & - & 3x_2 & - & 4x_3 & & & \\
\text{subject to} & & - & 2x_2 & - & 3x_3 & \geq & -5 & \text{(R1)} \\
& x_1 & + & x_2 & + & 2x_3 & \leq & 4 & \text{(R2)} \\
& x_1 & + & 2x_2 & + & 3x_3 & \leq & 7 & \text{(R3)} \\
& & & x_1, & & x_2, \; x_3 & \geq & 0 &
\end{array}
$$

The corresponding MPS-format SAS data set is as follows:

```
data example;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME          .       EXAMPLE    .     .      .
ROWS          .       .          .     .      .
N             COST    .          .     .      .
G             R1      .          .     .      .
L             R2      .          .     .      .
L             R3      .          .     .      .
COLUMNS       .       .          .     .      .
.             X1      COST       2     R2     1
.             X1      R3         1     .      .
.             X2      COST      -3     R1    -2
.             X2      R2         1     R3     2
.             X3      COST      -4     R1    -3
.             X3      R2         2     R3     3
RHS           .       .          .     .      .
.             RHS     R1        -5     R2     4
.             RHS     R3         7     .      .
ENDATA        .       .          .     .      .
;
```

You can also create this data set from an MPS-format flat file (examp.mps) by using the following SAS macro:

```
%mps2sasd(mpsfile = "examp.mps", outdata = example);
```

NOTE: The SAS macro %MPS2SASD is provided in SAS/OR software. See "Converting an MPS/QPS-Format File: %MPS2SASD" on page 848 for details.

You can use the following statement to call the OPTLP procedure:

```
title1 'The OPTLP Procedure';
proc optlp data = example
  objsense  = min
  presolver = automatic
  algorithm = primal
  primalout = expout
  dualout   = exdout;
run;
```

NOTE: The "N" designation for "COST" in the rows section of the data set example also specifies a minimization problem. See the section "ROWS Section" on page 841 for details.

The optimal primal and dual solutions are stored in the data sets expout and exdout, respectively, and are displayed in Figure 12.1.

```
title2 'Primal Solution';
proc print data=expout label;
run;

title2 'Dual Solution';
proc print data=exdout label;
run;
```

Figure 12.1 Primal and Dual Solution Output

The OPTLP Procedure
Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status	Reduced Cost
1	COST	RHS	X1	N	2	0	1.7977E308	0.0	L	2.0
2	COST	RHS	X2	N	-3	0	1.7977E308	2.5	B	0.0
3	COST	RHS	X3	N	-4	0	1.7977E308	0.0	L	0.5

The OPTLP Procedure
Dual Solution

Obs	Objective Function ID	RHS ID	Constraint Name	Constraint Type	Constraint RHS	Constraint Lower Bound	Constraint Upper Bound	Dual Variable Value	Constraint Status	Constraint Activity
1	COST	RHS	R1	G	-5	.	.	1.5	U	-5.0
2	COST	RHS	R2	L	4	.	.	0.0	B	2.5
3	COST	RHS	R3	L	7	.	.	0.0	B	5.0

For details about the type and status codes displayed for variables and constraints, see the section "Data Input and Output" on page 582.

Syntax: OPTLP Procedure

The following statements are available in the OPTLP procedure:

PROC OPTLP < *options* > **;**
DECOMP < *options* > **;**
DECOMP_MASTER < *options* > **;**
DECOMP_SUBPROB < *options* > **;**
PERFORMANCE < *performance-options* > **;**

Functional Summary

Table 12.1 summarizes the list of options available for the OPTLP procedure, classified by function.

Table 12.1 Options for the OPTLP Procedure

Description	Option
Data Set Options	
Specifies the input data set	DATA=
Specifies the dual input data set for warm start	DUALIN=
Specifies the dual solution output data set	DUALOUT=
Specifies whether the LP model is a maximization or minimization problem	OBJSENSE=
Specifies the primal input data set for warm start	PRIMALIN=
Specifies the primal solution output data set	PRIMALOUT=
Saves output data sets only if optimal	SAVE_ONLY_IF_OPTIMAL
Solver Options	
Enables or disables IIS detection	IIS=
Specifies the type of algorithm	ALGORITHM=
Specifies the type of algorithm called after network simplex	ALGORITHM2=
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Controls the dualization of the problem	DUALIZE=
Control Options	
Specifies the feasibility tolerance	FEASTOL=
Specifies the frequency of printing solution progress	LOGFREQ=
Specifies the detail of solution progress printed in log	LOGLEVEL=
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the optimality tolerance	OPTTOL=
Enables or disables printing summary	PRINTLEVEL=
Specifies units of CPU time or real time	TIMETYPE=
Simplex Algorithm Options	
Specifies the type of initial basis	BASIS=
Specifies the type of pricing strategy	PRICETYPE=
Specifies the queue size for pricing	QUEUESIZE=
Enables or disables scaling of the problem	SCALE=
Specifies the initial seed for the random number generator	SEED=
Interior Point Algorithm Options	
Enables or disables interior crossover	CROSSOVER=
Specifies the stopping criterion based on duality gap	STOP_DG=
Specifies the stopping criterion based on dual infeasibility	STOP_DI=
Specifies the stopping criterion based on primal infeasibility	STOP_PI=

PROC OPTLP Statement

PROC OPTLP < *options* > **;**

You can specify the following options in the PROC OPTLP statement.

Data Set Options

DATA=*SAS-data-set*
specifies the input data set corresponding to the LP model. If this option is not specified, PROC OPTLP will use the most recently created SAS data set. See Chapter 17, "The MPS-Format SAS Data Set," for more details about the input data set.

DUALIN=*SAS-data-set*

DIN=*SAS-data-set*
specifies the input data set corresponding to the dual solution that is required for warm starting the primal and dual simplex algorithms. See the section "Data Input and Output" on page 582 for details.

DUALOUT=*SAS-data-set*

DOUT=*SAS-data-set*
specifies the output data set for the dual solution. This data set contains the dual solution information. See the section "Data Input and Output" on page 582 for details.

OBJSENSE=*option*
specifies whether the LP model is a minimization or a maximization problem. You specify OBJSENSE=MIN for a minimization problem and OBJSENSE=MAX for a maximization problem. Alternatively, you can specify the objective sense in the input data set; see the section "ROWS Section" on page 841 for details. If for some reason the objective sense is specified differently in these two places, this option supersedes the objective sense specified in the input data set. If the objective sense is not specified anywhere, then PROC OPTLP interprets and solves the linear program as a minimization problem.

PRIMALIN=*SAS-data-set*

PIN=*SAS-data-set*
specifies the input data set corresponding to the primal solution that is required for warm starting the primal and dual simplex algorithms. See the section "Data Input and Output" on page 582 for details.

PRIMALOUT=*SAS-data-set*

POUT=*SAS-data-set*
specifies the output data set for the primal solution. This data set contains the primal solution information. See the section "Data Input and Output" on page 582 for details.

SAVE_ONLY_IF_OPTIMAL
specifies that the PRIMALOUT= and DUALOUT= data sets be saved only if the final solution obtained by the solver at termination is optimal. If the PRIMALOUT= and DUALOUT= options are specified, then by default (that is, omitting the SAVE_ONLY_IF_OPTIMAL option), PROC OPTLP always saves the solutions obtained at termination, regardless of the final status. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the output data sets can contain an intermediate solution, if one is available.

Solver Options

IIS=*number* | *string*

specifies whether PROC OPTLP attempts to identify a set of constraints and variables that form an irreducible infeasible set (IIS). Table 12.2 describes the valid values of the IIS= option.

Table 12.2 Values for IIS= Option

number	*string*	**Description**
0	OFF	Disables IIS detection.
1	ON	Enables IIS detection.

If an IIS is found, information about infeasible constraints or variable bounds can be found in the DUALOUT= and PRIMALOUT= data sets. The default value of this option is OFF. See the section "Irreducible Infeasible Set" on page 597 for details.

ALGORITHM=*option*

SOLVER=*option*

SOL=*option*

specifies one of the following LP algorithms:

Option	**Description**
PRIMAL (PS)	Uses primal simplex algorithm.
DUAL (DS)	Uses dual simplex algorithm.
NETWORK (NS)	Uses network simplex algorithm.
INTERIORPOINT (IP)	Uses interior point algorithm.
CONCURRENT (CON)	Uses several different algorithms in parallel.

The valid abbreviated value for each option is indicated in parentheses. By default, the dual simplex algorithm is used.

ALGORITHM2=*option*

SOLVER2=*option*

specifies one of the following LP algorithms if ALGORITHM=NS:

Option	**Description**
PRIMAL (PS)	Uses primal simplex algorithm (after network simplex).
DUAL (DS)	Uses dual simplex algorithm (after network simplex).

The valid abbreviated value for each option is indicated in parentheses. By default, the OPTLP procedure decides which algorithm is best to use after calling the network simplex algorithm on the extracted network.

Presolve Options

PRESOLVER=*number* | *string*

PRESOL=*number* | *string*

specifies one of the following presolve options:

number	*string*	**Description**
–1	AUTOMATIC	Applies presolver by using default settings.
0	NONE	Disables presolver.
1	BASIC	Performs basic presolve such as removing empty rows, columns, and fixed variables.
2	MODERATE	Performs basic presolve and applies other inexpensive presolve techniques.
3	AGGRESSIVE	Performs moderate presolve and applies other aggressive (but expensive) presolve techniques.

The default option is AUTOMATIC (–1), which is somewhere between the MODERATE and AGRESSIVE setting. See the section "Presolve" on page 586 for details.

DUALIZE=*number* | *string*

controls the dualization of the problem:

number	*string*	**Description**
–1	AUTOMATIC	The presolver uses a heuristic to decide whether to dualize the problem or not.
0	OFF	Disables dualization. The optimization problem is solved in the form that you specify.
1	ON	The presolver formulates the dual of the linear optimization problem.

Dualization is usually helpful for problems that have many more constraints than variables. You can use this option with all simplex algorithms in PROC OPTLP, but it is most effective with the primal and dual simplex algorithms.

The default option is AUTOMATIC.

Control Options

FEASTOL=ϵ

specifies the feasibility tolerance $\epsilon \in$[1E–9, 1E–4] for determining the feasibility of a variable value. The default value is 1E–6.

LOGFREQ=*k*

PRINTFREQ=*k*

specifies that the printing of the solution progress to the iteration log is to occur after every k iterations. The print frequency, k, is an integer between zero and the largest four-byte signed integer, which is $2^{31} - 1$.

The value $k = 0$ disables the printing of the progress of the solution.

If the LOGFREQ= option is not specified, then PROC OPTLP displays the iteration log with a dynamic frequency according to the problem size if the primal or dual simplex algorithm is used, with frequency 10,000 if the network simplex algorithm is used, or with frequency 1 if the interior point algorithm is used.

LOGLEVEL=*number* | *string*

PRINTLEVEL2=*number* | *string*

controls the amount of information displayed in the SAS log by the LP solver, from a short description of presolve information and summary to details at each iteration. Table 12.7 describes the valid values for this option.

Table 12.7 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turn off all solver-related messages in SAS log.
1	BASIC	Display a solver summary after stopping.
2	MODERATE	Print a solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.
3	AGGRESSIVE	Print a detailed solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.

The default value is MODERATE.

MAXITER=*k*

specifies the maximum number of iterations. The value k can be any integer between one and the largest four-byte signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed. For network simplex, this iteration limit corresponds to the algorithm called after network simplex (either primal or dual simplex).

MAXTIME=*t*

specifies an upper limit of t seconds of time for reading in the data and performing the optimization process. The value of the TIMETYPE= option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of t can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OPTTOL=ϵ

specifies the optimality tolerance $\epsilon \in$[1E–9, 1E–4] for declaring optimality. The default value is 1E–6.

PRINTLEVEL=0 | **1** | **2**

specifies whether a summary of the problem and solution should be printed. If PRINTLEVEL=1, then the Output Delivery System (ODS) tables ProblemSummary, SolutionSummary, and PerformanceInfo are produced and printed. If PRINTLEVEL=2, then the same tables are produced and printed along with an additional table called ProblemStatistics. If PRINTLEVEL=0, then no ODS tables are produced or printed. The default value is 1.

For details about the ODS tables created by PROC OPTLP, see the section "ODS Tables" on page 593.

TIMETYPE=*number* | *string*
specifies whether CPU time or real time is used for the MAXTIME= option and the _OROPTLP_ macro variable in a PROC OPTLP call. Table 12.8 describes the valid values of the TIMETYPE= option.

Table 12.8 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time.
1	REAL	Specifies units of real time.

The default value of the TIMETYPE= option depends on the values of the NTHREADS= and NODES= options in the PERFORMANCE statement. For more information about the NTHREADS= and NODES= options, see the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

If you specify a value greater than 1 for either the NTHREADS= or the NODES= option, the default value of the TIMETYPE= option is REAL. If you specify a value of 1 for both the NTHREADS= and NODES= options, the default value of the TIMETYPE= option is CPU.

Simplex Algorithm Options

BASIS=*number* | *string*
specifies the following options for generating an initial basis:

number	*string*	**Description**
0	CRASH	Generate an initial basis by using crash techniques (Maros 2003). The procedure creates a triangular basic matrix consisting of both decision variables and slack variables.
1	SLACK	Generate an initial basis by using all slack variables.
2	WARMSTART	Start the primal and dual simplex algorithms with a user-specified initial basis. The PRIMALIN= and DUALIN= data sets are required to specify an initial basis.

The default option is determined automatically based on the problem structure. For network simplex, this option has no effect.

PRICETYPE=*number* | *string*
specifies one of the following pricing strategies for the primal and dual simplex algorithms:

number	*string*	**Description**
0	HYBRID	Use a hybrid of Devex and steepest-edge pricing strategies. Available for the primal simplex algorithm only.
1	PARTIAL	Use Dantzig's rule on a queue of decision variables. Optionally, you can specify QUEUESIZE=. Available for the primal simplex algorithm only.
2	FULL	Use Dantzig's rule on all decision variables.
3	DEVEX	Use Devex pricing strategy.
4	STEEPESTEDGE	Use steepest-edge pricing strategy.

The default option is determined automatically based on the problem structure. For the network simplex algorithm, this option applies only to the algorithm specified by the ALGORITHM2= option. See the section "Pricing Strategies for the Primal and Dual Simplex Algorithms" on page 587 for details.

QUEUESIZE=*k*

specifies the queue size $k \in [1, n]$, where n is the number of decision variables. This queue is used for finding an entering variable in the simplex iteration. The default value is chosen adaptively based on the number of decision variables. This option is used only when PRICETYPE=PARTIAL.

SCALE=*number* | *string*

specifies one of the following scaling options:

number	*string*	**Description**
0	NONE	Disable scaling.
–1	AUTOMATIC	Automatically apply scaling procedure if necessary.

The default option is AUTOMATIC.

SEED=*number*

specifies the initial seed for the random number generator. Because the seed affects the perturbation in the simplex algorithms, the result might be a different optimal solution and a different solver path, but the effect is usually negligible. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. By default, SEED=100.

Interior Point Algorithm Options

CROSSOVER=*number* | *string*

specifies whether to convert the interior point solution to a basic simplex solution. If the interior point algorithm terminates with a solution, the crossover algorithm uses the interior point solution to create an initial basic solution. After performing primal fixing and dual fixing, the crossover algorithm calls a simplex algorithm to locate an optimal basic solution.

number	*string*	**Description**
0	OFF	Do not convert the interior point solution to a basic simplex solution.
1	ON	Convert the interior point solution to a basic simplex solution.

The default value of the CROSSOVER= option is ON.

STOP_DG=δ

specifies the desired relative duality gap $\delta \in$[1E–9, 1E–4]. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 588 for details.

STOP_DI=β

specifies the maximum allowed relative dual constraints violation $\beta \in$[1E–9, 1E–4]. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 588 for details.

STOP_PI=α
specifies the maximum allowed relative bound and primal constraints violation $\alpha \in [1E{-}9, 1E{-}4]$. The default value is 1E–6. See the section "The Interior Point Algorithm" on page 588 for details.

Decomposition Algorithm Statements

The following statements are available for the decomposition algorithm in the OPTLP procedure:

DECOMP < *options* > **;**

DECOMP_MASTER < *options* > **;**

DECOMP_SUBPROB < *options* > **;**

For more information about these statements, see Chapter 15, "The Decomposition Algorithm."

PERFORMANCE Statement

PERFORMANCE < *performance-options* > **;**

The PERFORMANCE statement specifies *performance-options* for single-machine mode and distributed mode, and requests detailed performance results of the OPTLP procedure.

You can also use the PERFORMANCE statement to control whether the OPTLP procedure executes in single-machine or distributed mode. The PERFORMANCE statement is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

For the OPTLP procedure, the decomposition algorithm, interior point algorithm, and concurrent LP algorithm can be run in single-machine mode. Only the decomposition algorithm can be run in distributed mode. The decomposition algorithm and concurrent LP algorithm support both the deterministic and nondeterministic modes. The interior point algorithm supports only the deterministic mode.

NOTE: Distributed mode requires SAS High-Performance Optimization.

Details: OPTLP Procedure

Data Input and Output

This subsection describes the PRIMALIN= and DUALIN= data sets required to warm start the primal and dual simplex algorithms, and the PRIMALOUT= and DUALOUT= output data sets.

Definitions of Variables in the PRIMALIN= Data Set

The PRIMALIN= data set has two required variables defined as follows:

VAR
specifies the name of the decision variable.

STATUS
specifies the status of the decision variable. It can take one of the following values:

B basic variable

L nonbasic variable at its lower bound

U nonbasic variable at its upper bound

F free variable

A newly added variable in the modified LP model when using the BASIS=WARMSTART option

NOTE: The PRIMALIN= data set is created from the PRIMALOUT= data set that is obtained from a previous "normal" run of PROC OPTLP (one that uses only the DATA= data set as the input).

Definitions of Variables in the DUALIN= Data Set

The DUALIN= data set also has two required variables defined as follows:

ROW
specifies the name of the constraint.

STATUS
specifies the status of the slack variable for a given constraint. It can take one of the following values:

B basic variable

L nonbasic variable at its lower bound

U nonbasic variable at its upper bound

F free variable

A newly added variable in the modified LP model when using the BASIS=WARMSTART option

NOTE: The DUALIN= data set is created from the DUALOUT= data set that is obtained from a previous "normal" run of PROC OPTLP (one that uses only the DATA= data set as the input).

Definitions of Variables in the PRIMALOUT= Data Set

The PRIMALOUT= data set contains the primal solution to the LP model; each observation corresponds to a variable of the LP problem. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the PRIMALOUT= data set can contain an intermediate solution, if one is available. See Example 12.1 for an example of the PRIMALOUT= data set. The variables in the data set have the following names and meanings.

_OBJ_ID_
specifies the name of the objective function. This is particularly useful when there are multiple objective functions, in which case each objective function has a unique name.

NOTE: PROC OPTLP does not support simultaneous optimization of multiple objective functions in this release.

_RHS_ID_
specifies the name of the variable that contains the right-hand-side value of each constraint.

VAR
specifies the name of the decision variable.

TYPE
specifies the type of the decision variable. _TYPE_ can take one of the following values:

N nonnegative

D bounded (with both lower and upper bound)

F free

X fixed

O other (with either lower or upper bound)

OBJCOEF
specifies the coefficient of the decision variable in the objective function.

LBOUND
specifies the lower bound on the decision variable.

UBOUND
specifies the upper bound on the decision variable.

VALUE
specifies the value of the decision variable.

STATUS
specifies the status of the decision variable. _STATUS_ can take one of the following values:

B basic variable

L nonbasic variable at its lower bound

U nonbasic variable at its upper bound

F free variable

A superbasic variable (a nonbasic variable that has a value strictly between its bounds)

I LP model infeasible (all decision variables have _STATUS_ equal to I)

For the interior point algorithm with IIS= OFF, _STATUS_ is blank.

The following values can appear only if IIS= ON. See the section "Irreducible Infeasible Set" on page 597 for details.

I_L the lower bound of the variable is needed for the IIS

I_U the upper bound of the variable is needed for the IIS

I_F both bounds of the variable needed for the IIS (the variable is fixed or has conflicting bounds)

_R_COST_
specifies the reduced cost of the decision variable, which is the amount by which the objective function is increased per unit increase in the decision variable. The reduced cost associated with the *i*th variable is the *i*th entry of the following vector:

$$\left(\mathbf{c}^{\mathrm{T}} - \mathbf{c}_{\boldsymbol{B}}^{\mathrm{T}}\mathbf{B}^{-1}\mathbf{A}\right)$$

where $\mathbf{B} \in \mathbb{R}^{m \times m}$ denotes the basis (matrix composed of *basic* columns of the constraints matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$), $\mathbf{c} \in \mathbb{R}^{n}$ is the vector of objective function coefficients, and $\mathbf{c}_{\boldsymbol{B}} \in \mathbb{R}^{m}$ is the vector of objective coefficients of the variables in the basis.

Definitions of Variables in the DUALOUT= Data Set

The DUALOUT= data set contains the dual solution to the LP model; each observation corresponds to a constraint of the LP problem. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the PRIMALOUT= data set can contain an intermediate solution, if one is available. Information about the objective rows of the LP problems is not included. See Example 12.1 for an example of the DUALOUT= data set. The variables in the data set have the following names and meanings.

_OBJ_ID_
specifies the name of the objective function. This is particularly useful when there are multiple objective functions, in which case each objective function has a unique name.

NOTE: PROC OPTLP does not support simultaneous optimization of multiple objective functions in this release.

_RHS_ID_
specifies the name of the variable that contains the right-hand-side value of each constraint.

ROW
specifies the name of the constraint.

TYPE
specifies the type of the constraint. _TYPE_ can take one of the following values:

L "less than or equals" constraint

E equality constraint

G "greater than or equals" constraint

R ranged constraint (both "less than or equals" and "greater than or equals")

RHS
specifies the value of the right-hand side of the constraint. It takes a missing value for a ranged constraint.

_L_RHS_
specifies the lower bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_U_RHS_
specifies the upper bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

VALUE
specifies the value of the dual variable associated with the constraint.

STATUS
specifies the status of the slack variable for the constraint. _STATUS_ can take one of the following values:

B basic variable

L nonbasic variable at its lower bound

U nonbasic variable at its upper bound

F free variable

A superbasic variable (a nonbasic variable that has a value strictly between its bounds)

I LP model infeasible (all decision variables have _STATUS_ equal to I)

The following values can appear only if option IIS= ON. See the section "Irreducible Infeasible Set" on page 597 for details.

I_L the "GE" ($\geq$) condition of the constraint is needed for the IIS

I_U the "LE" ($\leq$) condition of the constraint is needed for the IIS

I_F both conditions of the constraint are needed for the IIS (the constraint is an equality or a range constraint with conflicting bounds)

ACTIVITY
specifies the left-hand-side value of a constraint. In other words, the value of _ACTIVITY_ for the ith constraint would be equal to $\mathbf{a}_i^{\mathrm{T}}\mathbf{x}$, where $\mathbf{a}_i$ refers to the ith row of the constraints matrix and $\mathbf{x}$ denotes the vector of current decision variable values.

Presolve

Presolve in PROC OPTLP uses a variety of techniques to reduce the problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels or disable it by specifying the PRESOLVER= option.

Pricing Strategies for the Primal and Dual Simplex Algorithms

Several pricing strategies for the primal and dual simplex algorithms are available. Pricing strategies determine which variable enters the basis at each simplex pivot. They can be controlled by specifying the PRICETYPE= option.

The primal simplex algorithm has the following five pricing strategies:

PARTIAL	uses Dantzig's most violated reduced cost rule (Dantzig 1963). It scans a queue of decision variables and selects the variable with the most violated reduced cost as the entering variable. You can optionally specify the QUEUESIZE= option to control the length of this queue.
FULL	uses Dantzig's most violated reduced cost rule. It compares the reduced costs of all decision variables and selects the variable with the most violated reduced cost as the entering variable.
DEVEX	implements the Devex pricing strategy developed by Harris (1973).
STEEPESTEDGE	uses the steepest-edge pricing strategy developed by Forrest and Goldfarb (1992).
HYBRID	uses a hybrid of the Devex and steepest-edge pricing strategies.

The dual simplex algorithm has only three pricing strategies available: FULL, DEVEX, and STEEPEST-EDGE.

Warm Start for the Primal and Dual Simplex Algorithms

You can warm start the primal and dual simplex algorithms by specifying the option BASIS=WARMSTART. Additionally you need to specify the PRIMALIN= and DUALIN= data sets. The primal and dual simplex algorithms start with the basis thus provided. If the given basis cannot form a valid basis, the algorithms use the basis generated using their *crash* techniques.

After an LP model is solved using the primal and dual simplex algorithms, the BASIS=WARMSTART option enables you to perform sensitivity analysis such as modifying the objective function, changing the right-hand sides of the constraints, adding or deleting constraints or decision variables, and combinations of these cases. A faster solution to such a modified LP model can be obtained by starting with the basis in the optimal solution to the original LP model. This can be done by using the BASIS=WARMSTART option, modifying the DATA= input data set, and specifying the PRIMALIN= and DUALIN= data sets. Example 12.4 and Example 12.5 illustrate how to reoptimize an LP problem with a modified objective function and a modified right-hand side by using this technique. Example 12.6 shows how to reoptimize an LP problem after adding a new constraint.

The network simplex algorithm ignores the option BASIS=WARMSTART.

CAUTION: Since the presolver uses the objective function and/or right-hand-side information, the basis provided by you might not be valid for the presolved model. It is therefore recommended that you turn the PRESOLVER= option off when using BASIS=WARMSTART.

The Network Simplex Algorithm

The network simplex algorithm in PROC OPTLP attempts to leverage the speed of the network simplex algorithm to more efficiently solve linear programs by using the following process:

1. It heuristically extracts the largest possible network substructure from the original problem.
2. It uses the network simplex algorithm to solve for an optimal solution to this substructure.
3. It uses this solution to construct an advanced basis to warm-start either the primal or dual simplex algorithm on the original linear programming problem.

The network simplex algorithm is a specialized version of the simplex algorithm that uses spanning-tree bases to more efficiently solve linear programming problems that have a pure network form. Such LPs can be modeled using a formulation over a directed graph, as a minimum-cost flow problem. Let $G = (N, A)$ be a directed graph, where N denotes the nodes and A denotes the arcs of the graph. The decision variable x_{ij} denotes the amount of flow sent from node i to node j. The cost per unit of flow on the arcs is designated by c_{ij}, and the amount of flow sent across each arc is bounded to be within $[l_{ij}, u_{ij}]$. The demand (or supply) at each node is designated as b_i, where $b_i > 0$ denotes a supply node and $b_i < 0$ denotes a demand node. The corresponding linear programming problem is as follows:

$$\begin{array}{rlcll} \min & \sum_{(i,j)\in A} c_{ij}x_{ij} & & & \\ \text{subject to} & \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} & = & b_i & \forall i \in N \\ & x_{ij} & \leq & u_{ij} & \forall (i,j) \in A \\ & x_{ij} & \geq & l_{ij} & \forall (i,j) \in A \end{array}$$

The network simplex algorithm used in PROC OPTLP is the primal network simplex algorithm. This algorithm finds the optimal primal feasible solution and a dual solution that satisfies complementary slackness. Sometimes the directed graph G is disconnected. In this case, the problem can be decomposed into its weakly connected components and each minimum-cost flow problem can be solved separately. After solving each component, the optimal basis for the network substructure is augmented with the non-network variables and constraints from the original problem. This advanced basis is then used as a starting point for the primal or dual simplex method. The solver automatically selects the algorithm to use after network simplex. However, you can override this selection with the ALGORITHM2= option.

The network simplex algorithm can be more efficient than the other algorithms on problems with a large network substructure. You can view the size of the network structure in the log.

The Interior Point Algorithm

The interior point algorithm in PROC OPTLP implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following LP formulation (the primal):

$$\begin{array}{rl} \min & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

The corresponding dual formulation is as follows:

$$\begin{array}{rlcrcl} \max & \mathbf{b}^{\mathrm{T}}\mathbf{y} & & & & \\ \text{subject to} & \mathbf{A}^{\mathrm{T}}\mathbf{y} & + & \mathbf{w} & = & \mathbf{c} \\ & & & \mathbf{y} & \geq & \mathbf{0} \\ & & & \mathbf{w} & \geq & \mathbf{0} \end{array}$$

where $\mathbf{y} \in \mathbb{R}^m$ refers to the vector of dual variables and $\mathbf{w} \in \mathbb{R}^n$ refers to the vector of dual slack variables.

The dual formulation makes an important contribution to the certificate of optimality for the primal formulation. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows:

$$\begin{array}{rcll} \mathbf{Ax} - \mathbf{s} & = & \mathbf{b} & (\text{primal feasibility}) \\ \mathbf{A}^{\mathrm{T}}\mathbf{y} + \mathbf{w} & = & \mathbf{c} & (\text{dual feasibility}) \\ \mathbf{WXe} & = & \mathbf{0} & (\text{complementarity}) \\ \mathbf{SYe} & = & \mathbf{0} & (\text{complementarity}) \\ \mathbf{x},\ \mathbf{y},\ \mathbf{w},\ \mathbf{s} & \geq & \mathbf{0} & \end{array}$$

where $\mathbf{e} \equiv (1, \ldots, 1)^{\mathrm{T}}$ of appropriate dimension and $\mathbf{s} \in \mathbb{R}^m$ is the vector of primal *slack* variables.

NOTE: Slack variables (the *s* vector) are automatically introduced by the algorithm when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the algorithm to handle slack variables much more efficiently.

The letters $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{W}$, and $\mathbf{S}$ denote matrices with corresponding *x*, *y*, *w*, and *s* on the main diagonal and zero elsewhere, as in the following example:

$$\mathbf{X} \equiv \begin{bmatrix} x_1 & 0 & \cdots & 0 \\ 0 & x_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x_n \end{bmatrix}$$

If $(\mathbf{x}^*, \mathbf{y}^*, \mathbf{w}^*, \mathbf{s}^*)$ is a solution of the previously defined system of equations that represent the KKT conditions, then $\mathbf{x}^*$ is also an optimal solution to the original LP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations,

$$\begin{bmatrix} \mathbf{Y}^{-1}\mathbf{S} & \mathbf{A} \\ \mathbf{A}^{\mathrm{T}} & -\mathbf{X}^{-1}\mathbf{W} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{y} \\ \Delta\mathbf{x} \end{bmatrix} = \begin{bmatrix} \Xi \\ \Theta \end{bmatrix}$$

where $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ denote the vector of *search directions* in the primal and dual spaces, respectively, and Θ and Ξ constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. PROC OPTLP uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point algorithm is that it takes full advantage of the sparsity in the constraint matrix, thereby enabling it to efficiently solve large-scale linear programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore, it is of interest to observe the following four measures where $\|v\|_2$ is the Euclidean norm of the vector *v*:

- relative primal infeasibility measure α:

$$\alpha = \frac{\|\mathbf{Ax} - \mathbf{b} - \mathbf{s}\|_2}{\|\mathbf{b}\|_2 + 1}$$

- relative dual infeasibility measure β:

$$\beta = \frac{\|\mathbf{c} - \mathbf{A}^{\mathrm{T}}\mathbf{y} - \mathbf{w}\|_2}{\|\mathbf{c}\|_2 + 1}$$

- relative duality gap δ:

$$\delta = \frac{|\mathbf{c}^{\mathrm{T}}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{y}|}{|\mathbf{c}^{\mathrm{T}}\mathbf{x}| + 1}$$

- absolute complementarity γ:

$$\gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i$$

These measures are displayed in the iteration log.

Iteration Log for the Primal and Dual Simplex Algorithms

The primal and dual simplex algorithms implement a two-phase simplex algorithm. Phase I finds a feasible solution, which phase II improves to an optimal solution.

When LOGFREQ=1, the following information is printed in the iteration log:

Algorithm	indicates which simplex method is running by printing the letter P (primal) or D (dual).
Phase	indicates whether the algorithm is in phase I or phase II of the simplex method.
Iteration	indicates the iteration number.
Objective Value	indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
Time	indicates the time elapsed (in seconds).
Entering Variable	indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by "(S)". If the entering nonbasic variable has distinct and finite lower and upper bounds, then a "bound swap" can take place in the primal simplex method.
Leaving Variable	indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by "(S)". The leaving variable is the same as the entering variable if a bound swap has taken place.

When you omit the LOGFREQ= option or specify a value greater than 1, only the algorithm, phase, iteration, objective value, and time information is printed in the iteration log.

The behavior of objective values in the iteration log depends on both the current phase and the chosen algorithm. In phase I, both simplex methods have artificial objective values that decrease to 0 when a feasible solution is found. For the dual simplex method, phase II maintains a dual feasible solution, so a minimization problem has increasing objective values in the iteration log. For the primal simplex method, phase II maintains a primal feasible solution, so a minimization problem has decreasing objective values in the iteration log.

During the solution process, some elements of the LP model might be perturbed to improve performance. In this case the objective values that are printed correspond to the perturbed problem. After reaching optimality for the perturbed problem, PROC OPTLP solves the original problem by switching from the primal simplex method to the dual simplex method (or from the dual to the primal simplex method). Because the problem might be perturbed again, this process can result in several changes between the two algorithms.

Iteration Log for the Network Simplex Algorithm

After finding the embedded network and formulating the appropriate relaxation, the network simplex algorithm uses a primal network simplex algorithm. In the case of a connected network, with one (weakly connected) component, the log shows the progress of the simplex algorithm. The following information is displayed in the iteration log:

Iteration	indicates the iteration number.
PrimalObj	indicates the primal objective value of the current solution.
Primal Infeas	indicates the maximum primal infeasibility of the current solution.
Time	indicates the time spent on the current component by network simplex.

The frequency of the simplex iteration log is controlled by the LOGFREQ= option. The default value of the LOGFREQ= option is 10,000.

If the network relaxation is disconnected, the information in the iteration log shows progress at the component level. The following information is displayed in the iteration log:

Component	indicates the component number being processed.
Nodes	indicates the number of nodes in this component.
Arcs	indicates the number of arcs in this component.
Iterations	indicates the number of simplex iterations needed to solve this component.
Time	indicates the time spent so far in network simplex.

The frequency of the component iteration log is controlled by the LOGFREQ= option. In this case, the default value of the LOGFREQ= option is determined by the size of the network.

The LOGLEVEL= option adjusts the amount of detail shown. By default, LOGLEVEL= is set to MODERATE and reports as described previously. If set to NONE, no information is shown. If set to BASIC, the only information shown is a summary of the network relaxation and the time spent solving the relaxation. If set to AGGRESSIVE, in the case of one component, the log displays as described previously; in the case of multiple components, for each component, a separate simplex iteration log is displayed.

Iteration Log for the Interior Point Algorithm

The interior point algorithm implements an infeasible primal-dual predictor-corrector interior point algorithm. The following information is displayed in the iteration log:

Iter	indicates the iteration number.
Complement	indicates the (absolute) complementarity.
Duality Gap	indicates the (relative) duality gap.
Primal Infeas	indicates the (relative) primal infeasibility measure.
Bound Infeas	indicates the (relative) bound infeasibility measure.
Dual Infeas	indicates the (relative) dual infeasibility measure.
Time	indicates the time elapsed (in seconds).

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. If they do not, it can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value specified in the MAXITER= or MAXTIME= options. If the complementarity or the duality gap do not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

Iteration Log for the Crossover Algorithm

The crossover algorithm takes an optimal solution from the interior point algorithm and transforms it into an optimal basic solution. The iterations of the crossover algorithm are similar to simplex iterations; this similarity is reflected in the format of the iteration logs.

When LOGFREQ=1, the following information is printed in the iteration log:

Phase	indicates whether the primal crossover (PC) or dual crossover (DC) technique is used.
Iteration	indicates the iteration number.
Objective Value	indicates the total amount by which the superbasic variables are off their bound. This value decreases to 0 as the crossover algorithm progresses.
Time	indicates the time elapsed (in seconds).
Entering Variable	indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by "(S)."
Leaving Variable	indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by "(S)."

When you omit the LOGFREQ= option or specify a value greater than 1, only the phase, iteration, objective value, and time information is printed in the iteration log.

After all the superbasic variables have been eliminated, the crossover algorithm continues with regular primal or dual simplex iterations.

Concurrent LP

The ALGORITHM=CON option starts several different linear optimization algorithms in parallel in a single-machine mode. The OPTLP procedure automatically determines which algorithms to run and how many threads to assign to each algorithm. If sufficient resources are available, the procedure runs all four standard algorithms. When the first algorithm ends, the procedure returns the results from that algorithm and terminates any other algorithms that are still running. If you specify a value of DETERMINISTIC for the PARALLELMODE= option in the PERFORMANCE statement, the algorithm for which the results are returned is not necessarily the one that finished first. The OPTLP procedure deterministically selects the algorithm for which the results are returned. Regardless of which mode (deterministic or nondeterministic) is in effect, terminating algorithms that are still running might take a significant amount of time.

During concurrent optimization, the procedure displays the iteration log for the dual simplex algorithm. For more information about this iteration log, see the section "Iteration Log for the Primal and Dual Simplex Algorithms" on page 590. Upon termination, the procedure displays the iteration log for the algorithm that finishes first, unless the dual simplex algorithm finishes first. If you specify LOGLEVEL=AGGRESSIVE, the OPTLP procedure displays the iteration logs for all algorithms that are run concurrently.

If you specify PRINTLEVEL=2 and ALGORITHM=CON, the OPTLP procedure produces an ODS table called ConcurrentSummary. This table contains a summary of the solution statuses of all algorithms that are run concurrently.

Parallel Processing

The interior point and concurrent LP algorithms can be run in single-machine mode (in single-machine mode, the computation is executed by multiple threads on a single computer). The decomposition algorithm can be run in either single-machine or distributed mode (in distributed mode, the computation is executed on multiple computing nodes in a distributed computing environment).

NOTE: Distributed mode requires SAS High-Performance Optimization.

You can specify options for parallel processing in the PERFORMANCE statement, which is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

ODS Tables

PROC OPTLP creates three Output Delivery System (ODS) tables by default. The first table, ProblemSummary, is a summary of the input LP problem. The second table, SolutionSummary, is a brief summary of the solution status. The third table, PerformanceInfo, is a summary of performance options. You can use ODS table names to select tables and create output data sets. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

If you specify a value of 2 for the PRINTLEVEL= option, then the ProblemStatistics table is produced. This table contains information about the problem data. For more information, see the section "Problem Statistics" on page 596. If you specify PRINTLEVEL=2 and ALGORITHM=CON, the ConcurrentSummary table is produced. This table contains solution status information for all algorithms that are run concurrently. For more information, see the section "Concurrent LP" on page 593.

If you specify the DETAILS option in the PERFORMANCE statement, then the Timing table is produced.

Table 12.13 lists all the ODS tables that can be produced by the OPTLP procedure, along with the statement and option specifications required to produce each table.

Table 12.13 ODS Tables Produced by PROC OPTLP

ODS Table Name	Description	Statement	Option
ProblemSummary	Summary of the input LP problem	PROC OPTLP	PRINTLEVEL=1 (default)
SolutionSummary	Summary of the solution status	PROC OPTLP	PRINTLEVEL=1 (default)
ProblemStatistics	Description of input problem data	PROC OPTLP	PRINTLEVEL=2
ConcurrentSummary	Summary of the solution status for all algorithms run concurrently	PROC OPTLP	PRINTLEVEL=2, ALGORITHM=CON
PerformanceInfo	List of performance options and their values	PROC OPTLP	PRINTLEVEL=1 (default)
Timing	Detailed solution timing	PERFORMANCE	DETAILS

A typical output of PROC OPTLP is shown in Figure 12.2.

Figure 12.2 Typical OPTLP Output

The OPTLP Procedure

Problem Summary	
Problem Name	ADLITTLE
Objective Sense	Minimization
Objective Function	.Z....
RHS	ZZZZ0001
Number of Variables	97
Bounded Above	0
Bounded Below	97
Bounded Above and Below	0
Free	0
Fixed	0
Number of Constraints	56
LE (<=)	40
EQ (=)	15
GE (>=)	1
Range	0
Constraint Coefficients	383

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Figure 12.2 *continued*

Solution Summary	
Solver	LP
Algorithm	Dual Simplex
Objective Function	.Z....
Solution Status	Optimal
Objective Value	225494.96316
Primal Infeasibility	3.502976E-12
Dual Infeasibility	1.546141E-11
Bound Infeasibility	0
Iterations	93
Presolve Time	0.00
Solution Time	0.02

You can create output data sets from these tables by using the ODS OUTPUT statement. This can be useful, for example, when you want to create a report to summarize multiple PROC OPTLP runs. The output data sets corresponding to the preceding output are shown in Figure 12.3, where you can also find (at the row following the heading of each data set in display) the variable names that are used in the table definition (template) of each table.

Figure 12.3 ODS Output Data Sets

Problem Summary

Obs	Label1	cValue1	nValue1
1	Problem Name	ADLITTLE	.
2	Objective Sense	Minimization	.
3	Objective Function	.Z....	.
4	RHS	ZZZZ0001	.
5			.
6	Number of Variables	97	97.000000
7	Bounded Above	0	0
8	Bounded Below	97	97.000000
9	Bounded Above and Below	0	0
10	Free	0	0
11	Fixed	0	0
12			.
13	Number of Constraints	56	56.000000
14	LE (<=)	40	40.000000
15	EQ (=)	15	15.000000
16	GE (>=)	1	1.000000
17	Range	0	0
18			.
19	Constraint Coefficients	383	383.000000

Figure 12.3 *continued*

Solution Summary

Obs	Label1	cValue1	nValue1
1	Solver	LP	.
2	Algorithm	Dual Simplex	.
3	Objective Function	.Z....	.
4	Solution Status	Optimal	.
5	Objective Value	225494.96316	225495
6			.
7	Primal Infeasibility	3.502976E-12	3.502976E-12
8	Dual Infeasibility	1.546141E-11	1.546141E-11
9	Bound Infeasibility	0	0
10			.
11	Iterations	93	93.000000
12	Presolve Time	0.00	0
13	Solution Time	0.02	0.015600

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTLP procedure causes the ODS table ProblemStatistics to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 12.4 demonstrates the contents of the ODS table ProblemStatistics.

Figure 12.4 ODS Table ProblemStatistics

The OPTLP Procedure

Problem Statistics	
Number of Constraint Matrix Nonzeros	8
Maximum Constraint Matrix Coefficient	3
Minimum Constraint Matrix Coefficient	1
Average Constraint Matrix Coefficient	1.875
Number of Objective Nonzeros	3
Maximum Objective Coefficient	4
Minimum Objective Coefficient	2
Average Objective Coefficient	3
Number of RHS Nonzeros	3
Maximum RHS	7
Minimum RHS	4
Average RHS	5.3333333333
Maximum Number of Nonzeros per Column	3
Minimum Number of Nonzeros per Column	2
Average Number of Nonzeros per Column	2.67
Maximum Number of Nonzeros per Row	3
Minimum Number of Nonzeros per Row	2
Average Number of Nonzeros per Row	2.67

Irreducible Infeasible Set

For a linear programming problem, an irreducible infeasible set (IIS) is an infeasible subset of constraints and variable bounds that will become feasible if any single constraint or variable bound is removed. It is possible to have more than one IIS in an infeasible LP. Identifying an IIS can help isolate the structural infeasibility in an LP.

The presolver in the OPTLP procedure can detect infeasibility, but it identifies only the variable bound or constraint that triggers the infeasibility.

The IIS=ON option directs the OPTLP procedure to search for an IIS in a specified LP. The OPTLP procedure does not apply the presolver to the problem during the IIS search. If PROC OPTLP detects an IIS, it first outputs the IIS to the data sets that are specified by the PRIMALOUT= and DUALOUT= options, and then it stops. The number of iterations that are reported in the macro variable and the ODS table is the total number of simplex iterations. This total includes the initial LP solve and all subsequent iterations during the constraint deletion phase.

The IIS= option can add special values to the _STATUS_ variables in the output data sets. (For more information, see the section "Data Input and Output" on page 582.) For constraints, a status of "I_L", "I_U", or "I_F" indicates that the "GE" (≥), "LE" (≤), or "EQ" (=) constraint, respectively, is part of the IIS. For range constraints, a status of "I_L" or "I_U" indicates that the lower or upper bound of the constraint,

respectively, is needed for the IIS, and "I_F" indicates that the bounds in the constraint are conflicting. For variables, a status of "I_L", "I_U", or "I_F" indicates that the lower, upper, or both bounds of the variable, respectively, are needed for the IIS. From this information, you can identify both the names of the constraints (variables) in the IIS and the corresponding bound where infeasibility occurs.

Making any one of the constraints or variable bounds in the IIS nonbinding removes the infeasibility from the IIS. In some cases, changing a right-hand side or bound by a finite amount removes the infeasibility. However, the only way to guarantee removal of the infeasibility is to set the appropriate right-hand side or bound to ∞ or $-\infty$. Because it is possible for an LP to have multiple irreducible infeasible sets, simply removing the infeasibility from one set might not make the entire problem feasible. To make the entire problem feasible, you can specify IIS=ON and rerun PROC OPTLP after removing the infeasibility from an IIS. Repeating this process until the LP solver no longer detects an IIS results in a feasible problem. This approach to infeasibility repair can produce different end problems depending on which right-hand sides and bounds you choose to relax.

Changing different constraints and bounds can require considerably different changes to the MPS-format SAS data set. For example, if you use the default lower bound of 0 for a variable but you want to relax the lower bound to $-\infty$, you might need to add an MI row to the BOUNDS section of the data set. For more information about changing variable and constraint bounds, see Chapter 17, "The MPS-Format SAS Data Set."

The IIS= option in PROC OPTLP uses two different methods to identify an IIS:

1. Based on the result of the initial solve, the *sensitivity filter* removes several constraints and variable bounds immediately while still maintaining infeasibility. This phase is quick and dramatically reduces the size of the IIS.
2. Next, the *deletion filter* removes each remaining constraint and variable bound one by one to check which of them are needed to obtain an infeasible system. This second phase is more time consuming, but it ensures that the IIS set that PROC OPTLP returns is indeed irreducible. The progress of the deletion filter is reported at regular intervals. Occasionally, the sensitivity filter might be called again during the deletion filter to improve performance.

See Example 12.7 for an example that demonstrates the use of the IIS= option in locating and removing infeasibilities. You can find more details about IIS algorithms in Chinneck (2008).

Macro Variable _OROPTLP_

The OPTLP procedure defines a macro variable named _OROPTLP_. This variable contains a character string that indicates the status of the OPTLP procedure upon termination. The various terms of the variable are interpreted as follows.

STATUS
indicates the solver status at termination. It can take one of the following values:

OK	The procedure terminated normally.
SYNTAX_ERROR	Incorrect syntax was used.
DATA_ERROR	The input data were inconsistent.

OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem occurred in reading or writing data.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM
indicates the algorithm that produces the solution data in the macro variable. This term appears only when STATUS=OK. It can take one of the following values:

PS	The primal simplex algorithm produced the solution data.
DS	The dual simplex algorithm produced the solution data.
NS	The network simplex algorithm produced the solution data.
IP	The interior point algorithm produced the solution data.
DECOMP	The decomposition algorithm produced the solution data.

When you run algorithms concurrently (ALGORITHM=CON), this term indicates which algorithm is the first to terminate.

SOLUTION_STATUS
indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
CONDITIONAL_OPTIMAL	The solution is optimal, but some infeasibilities (primal, dual or bound) exceed tolerances due to scaling or preprocessing.
FEASIBLE	The problem is feasible.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
ITERATION_LIMIT_REACHED	The maximum allowable number of iterations was reached.
TIME_LIMIT_REACHED	The solver reached its execution time limit.
ABORTED	The solver was interrupted externally.
FAILED	The solver failed to converge, possibly due to numerical issues.

OBJECTIVE
indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the primal constraints by the primal solution. For the interior point algorithm, this term indicates the relative violation of the primal constraints by the primal solution.

DUAL_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the dual constraints by the dual solution. For the interior point algorithm, this term indicates the relative violation of the dual constraints by the dual solution.

BOUND_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the lower or upper bounds (or both) by the primal solution. For the interior point algorithm, this term indicates the relative violation of the lower or upper bounds (or both) by the primal solution.

DUALITY_GAP
indicates the (relative) duality gap. This term appears only if the interior point algorithm is used.

COMPLEMENTARITY
indicates the (absolute) complementarity. This term appears only if the interior point algorithm is used.

ITERATIONS
indicates the number of iterations taken to solve the problem. When the network simplex algorithm is used, this term indicates the number of network simplex iterations taken to solve the network relaxation. When crossover is enabled, this term indicates the number of interior point iterations taken to solve the problem.

ITERATIONS2
indicates the number of simplex iterations performed by the secondary algorithm. In network simplex, the secondary algorithm is selected automatically, unless a value has been specified for the ALGORITHM2= option. When crossover is enabled, the secondary algorithm is selected automatically. This term appears only if the network simplex algorithm is used or if crossover is enabled.

PRESOLVE_TIME
indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTLP_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

Examples: OPTLP Procedure

Example 12.1: Oil Refinery Problem

Consider an oil refinery scenario. A step in refining crude oil into finished oil products involves a distillation process that splits crude into various streams. Suppose there are three types of crude available: Arabian light (a_l), Arabian heavy (a_h), and Brega (br). These crudes are distilled into light naphtha (na_l), intermediate naphtha (na_i), and heating oil (h_o). These in turn are blended into two types of jet fuel. Jet fuel j_1 is made up of 30% intermediate naphtha and 70% heating oil, and jet fuel j_2 is made up of 20% light naphtha and 80% heating oil. What amounts of the three crudes maximize the profit from producing jet fuel (j_1, j_2)? This problem can be formulated as the following linear program:

$$
\begin{array}{rrcrcrcr}
\max & -175\,\text{a\_l} - 165\,\text{a\_h} - 205\,\text{br} + 350\,\text{j\_1} + 350\,\text{j\_2} & & & & & & \\
\text{subject to} & & & & & & & \\
\text{(napha\_l)} & 0.035\,\text{a\_l} & + & 0.03\,\text{a\_h} & + & 0.045\,\text{br} & = & \text{na\_l} \\
\text{(napha\_i)} & 0.1\,\text{a\_l} & + & 0.075\,\text{a\_h} & + & 0.135\,\text{br} & = & \text{na\_i} \\
\text{(htg\_oil)} & 0.39\,\text{a\_l} & + & 0.3\,\text{a\_h} & + & 0.43\,\text{br} & = & \text{h\_o} \\
\text{(blend1)} & & & 0.3\,\text{j\_1} & & & \leq & \text{na\_i} \\
\text{(blend2)} & & & & & 0.2\,\text{j\_2} & \leq & \text{na\_l} \\
\text{(blend3)} & & & 0.7\,\text{j\_1} & + & 0.8\,\text{j\_2} & \leq & \text{h\_o} \\
 & \text{a\_l} & & & & & \leq & 110 \\
 & & & \text{a\_h} & & & \leq & 165 \\
 & & & & & \text{br} & \leq & 80 \\
 & \text{a\_l, a\_h, br, na\_1, na\_i, h\_o, j\_1, j\_2} & & & & & \geq & 0
\end{array}
$$

The constraints "blend1" and "blend2" ensure that j_1 and j_2 are made with the specified amounts of na_i and na_l, respectively. The constraint "blend3" is actually the reduced form of the following constraints:

$$
\begin{array}{rcrcl}
\text{h\_o1} & & & \geq & 0.7\,\text{j\_1} \\
 & & \text{h\_o2} & \geq & 0.8\,\text{j\_2} \\
\text{h\_o1} & + & \text{h\_o2} & \leq & \text{h\_o}
\end{array}
$$

where h_o1 and h_o2 are dummy variables.

You can use the following SAS code to create the input data set ex1:

```
data ex1;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME          .          EX1        .      .          .
ROWS          .          .          .      .          .
 N            profit     .          .      .          .
 E            napha_l    .          .      .          .
 E            napha_i    .          .      .          .
 E            htg_oil    .          .      .          .
 L            blend1     .          .      .          .
 L            blend2     .          .      .          .
 L            blend3     .          .      .          .
COLUMNS       .          .          .      .          .
.             a_l        profit     -175   napha_l    .035
.             a_l        napha_i    .100   htg_oil    .390
.             a_h        profit     -165   napha_l    .030
.             a_h        napha_i    .075   htg_oil    .300
.             br         profit     -205   napha_l    .045
.             br         napha_i    .135   htg_oil    .430
.             na_l       napha_l    -1     blend2     -1
.             na_i       napha_i    -1     blend1     -1
.             h_o        htg_oil    -1     blend3     -1
.             j_1        profit     350    blend1     .3
.             j_1        blend3     .7     .          .
.             j_2        profit     350    blend2     .2
```

```
.           j_2         blend3    .8     .          .
BOUNDS      .           .         .      .          .
UP          .           a_l       110    .          .
UP          .           a_h       165    .          .
UP          .           br        80     .          .
ENDATA      .           .         .      .          .
;
```

You can use the following call to PROC OPTLP to solve the LP problem:

```
proc optlp data=ex1
   objsense  = max
   algorithm = primal
   primalout = ex1pout
   dualout   = ex1dout
   logfreq   = 1;
run;
%put &_OROPTLP_;
```

Note that the OBJSENSE=MAX option is used to indicate that the objective function is to be maximized.

The primal and dual solutions are displayed in Output 12.1.1.

Output 12.1.1 Example 1: Primal and Dual Solution Output

Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status	Reduced Cost
1	profit		a_l	D	-175	0	110	110.000	U	10.2083
2	profit		a_h	D	-165	0	165	0.000	L	-22.8125
3	profit		br	D	-205	0	80	80.000	U	2.8125
4	profit		na_l	N	0	0	1.7977E308	7.450	B	0.0000
5	profit		na_i	N	0	0	1.7977E308	21.800	B	0.0000
6	profit		h_o	N	0	0	1.7977E308	77.300	B	0.0000
7	profit		j_1	N	350	0	1.7977E308	72.667	B	0.0000
8	profit		j_2	N	350	0	1.7977E308	33.042	B	0.0000

Dual Solution

Obs	Objective Function ID	RHS ID	Constraint Name	Constraint Type	Constraint RHS	Constraint Lower Bound	Constraint Upper Bound	Dual Variable Value	Constraint Status	Constraint Activity
1	profit		napha_l	E	0	.	.	0.000	L	0.00000
2	profit		napha_i	E	0	.	.	-145.833	U	0.00000
3	profit		htg_oil	E	0	.	.	-437.500	U	0.00000
4	profit		blend1	L	0	.	.	145.833	L	-0.00000
5	profit		blend2	L	0	.	.	0.000	B	-0.84167
6	profit		blend3	L	0	.	.	437.500	L	0.00000

The progress of the solution is printed to the log as follows.

Output 12.1.2 Log: Solution Progress

```
NOTE: The problem EX1 has 8 variables (0 free, 0 fixed).
NOTE: The problem has 6 constraints (3 LE, 3 EQ, 0 GE, 0 range).
NOTE: The problem has 19 constraint coefficients.
WARNING: The objective sense has been changed to maximization.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 3 variables and 3 constraints.
NOTE: The LP presolver removed 6 constraint coefficients.
NOTE: The presolved problem has 5 variables, 3 constraints, and 13 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.
                          Objective                Entering      Leaving
      Phase Iteration       Value         Time     Variable      Variable
       P 1          1    0.000000E+00          0
       P 2          2    0.000000E+00          0        j_1         blend1 (S)
       P 2          3    1.405640E-01          0        j_2         blend3 (S)
       P 2          4    1.454487E-01          0        a_l         blend2 (S)
       P 2          5    2.379819E-01          0         br            a_l
       P 2          6    1.202394E+03          0     blend2 (S)        br
       P 2          7    1.348074E+03          0
       D 2          8    1.347917E+03          0
NOTE: Optimal.
NOTE: Objective = 1347.9166667.
NOTE: The Primal Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.EX1POUT has 8 observations and 10 variables.
NOTE: The data set WORK.EX1DOUT has 6 observations and 10 variables.
```

Note that the %put statement immediately after the OPTLP procedure prints value of the macro variable _OROPTLP_ to the log as follows.

Output 12.1.3 Log: Value of the Macro Variable _OROPTLP_

```
STATUS=OK ALGORITHM=PS SOLUTION_STATUS=OPTIMAL OBJECTIVE=1347.9166667
PRIMAL_INFEASIBILITY=2.888315E-15 DUAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0
ITERATIONS=8 PRESOLVE_TIME=0.00 SOLUTION_TIME=0.00
```

The value briefly summarizes the status of the OPTLP procedure upon termination.

Example 12.2: Using the Interior Point Algorithm

You can also solve the oil refinery problem described in Example 12.1 by using the interior point algorithm. You can create the input data set from an external MPS-format flat file by using the SAS macro %MPS2SASD or SAS DATA step code, both of which are described in "Getting Started: OPTLP Procedure" on page 572. You can use the following SAS code to solve the problem:

```
proc optlp data=ex1
  objsense  = max
  algorithm = ip
  primalout = ex1ipout
  dualout   = ex1idout
  logfreq   = 1;
run;
```

The optimal solution is displayed in Output 12.2.1.

Output 12.2.1 Interior Point Algorithm: Primal Solution Output

Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status	Reduced Cost
1	profit		a_l	D	-175	0	110	110.000	U	10.2083
2	profit		a_h	D	-165	0	165	0.000	L	-22.8125
3	profit		br	D	-205	0	80	80.000	U	2.8125
4	profit		na_l	N	0	0	1.7977E308	7.450	B	0.0000
5	profit		na_i	N	0	0	1.7977E308	21.800	B	0.0000
6	profit		h_o	N	0	0	1.7977E308	77.300	B	0.0000
7	profit		j_1	N	350	0	1.7977E308	72.667	B	0.0000
8	profit		j_2	N	350	0	1.7977E308	33.042	B	-0.0000

The iteration log is displayed in Output 12.2.2.

Output 12.2.2 Log: Solution Progress

```
NOTE: The problem EX1 has 8 variables (0 free, 0 fixed).
NOTE: The problem has 6 constraints (3 LE, 3 EQ, 0 GE, 0 range).
NOTE: The problem has 19 constraint coefficients.
WARNING: The objective sense has been changed to maximization.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 3 variables and 3 constraints.
NOTE: The LP presolver removed 6 constraint coefficients.
NOTE: The presolved problem has 5 variables, 3 constraints, and 13 constraint
      coefficients.
NOTE: The LP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Interior Point algorithm is using up to 4 threads.
                                        Primal       Bound        Dual
      Iter  Complement Duality Gap      Infeas      Infeas      Infeas   Time
         0  8.5854E+01  1.9793E+01  3.0659E+00  1.9225E-02  1.4035E-01      0
         1  1.2776E+01  4.0585E+00  3.0659E-02  1.9225E-04  1.8805E-02      0
         2  3.1694E+00  5.7554E-01  2.4563E-03  1.5402E-05  5.4311E-03      0
         3  4.6729E-01  7.9027E-02  2.2921E-04  1.4373E-06  2.4468E-04      0
         4  5.7860E-03  9.6069E-04  3.2729E-06  2.0522E-08  2.8762E-06      0
         5  5.7862E-05  9.6045E-06  3.2731E-08  2.0524E-10  2.8763E-08      0
         6  1.5657E-04  1.1524E-07  9.8105E-11  2.8290E-12  8.2666E-08      0
         7  0.0000E+00  5.0568E-16  2.8899E-15  0.0000E+00  3.0766E-15      0
NOTE: The Interior Point solve time is 0.00 seconds.
NOTE: The CROSSOVER option is enabled.
NOTE: The crossover basis contains 0 primal and 0 dual superbasic variables.
                           Objective
      Phase Iteration        Value         Time
       P C          1    0.000000E+00         0
       P 2          2    1.347917E+03         0
       D 2          3    1.347917E+03         0
NOTE: The Crossover time is 0.00 seconds.
NOTE: Optimal.
NOTE: Objective = 1347.9166667.
NOTE: The data set WORK.EX1IPOUT has 8 observations and 10 variables.
NOTE: The data set WORK.EX1IDOUT has 6 observations and 10 variables.
```

Example 12.3: The Diet Problem

Consider the problem of diet optimization. There are six different foods: bread, milk, cheese, potato, fish, and yogurt. The cost and nutrition values per unit are displayed in Table 12.14.

Table 12.14 Cost and Nutrition Values

	Bread	Milk	Cheese	Potato	Fish	Yogurt
Cost	2.0	3.5	8.0	1.5	11.0	1.0
Protein, g	4.0	8.0	7.0	1.3	8.0	9.2

Table 12.14 (continued)

	Bread	Milk	Cheese	Potato	Fish	Yogurt
Fat, g	1.0	5.0	9.0	0.1	7.0	1.0
Carbohydrates, g	15.0	11.7	0.4	22.6	0.0	17.0
Calories	90	120	106	97	130	180

The objective is to find a minimum-cost diet that contains at least 300 calories, not more than 10 grams of protein, not less than 10 grams of carbohydrates, and not less than 8 grams of fat. In addition, the diet should contain at least 0.5 unit of fish and no more than 1 unit of milk.

You can use the following SAS code to create the MPS-format input data set:

```
data ex3;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME         .          EX3        .     .          .
ROWS         .          .          .     .          .
 N           diet       .          .     .          .
 G           calories   .          .     .          .
 L           protein    .          .     .          .
 G           fat        .          .     .          .
 G           carbs      .          .     .          .
COLUMNS      .          .          .     .          .
.            br         diet       2     calories   90
.            br         protein    4     fat        1
.            br         carbs      15    .          .
.            mi         diet       3.5   calories   120
.            mi         protein    8     fat        5
.            mi         carbs      11.7  .          .
.            ch         diet       8     calories   106
.            ch         protein    7     fat        9
.            ch         carbs      .4    .          .
.            po         diet       1.5   calories   97
.            po         protein    1.3   fat        .1
.            po         carbs      22.6  .          .
.            fi         diet       11    calories   130
.            fi         protein    8     fat        7
.            fi         carbs      0     .          .
.            yo         diet       1     calories   180
.            yo         protein    9.2   fat        1
.            yo         carbs      17    .          .
RHS          .          .          .     .          .
.            .          calories   300   protein    10
.            .          fat        8     carbs      10
BOUNDS       .          .          .     .          .
UP           .          mi         1     .          .
LO           .          fi         .5    .          .
ENDATA       .          .          .     .          .
;
```

You can solve the diet problem by using PROC OPTLP as follows:

```
proc optlp data=ex3
   presolver = none
   algorithm = ps
   primalout = ex3pout
   dualout   = ex3dout
   logfreq   = 1;
run;
```

The solution summary and the optimal primal solution are displayed in Output 12.3.1.

Output 12.3.1 Diet Problem: Solution Summary and Optimal Primal Solution

Solution Summary

Obs	Label1	cValue1	nValue1
1	Solver	LP	.
2	Algorithm	Primal Simplex	.
3	Objective Function	diet	.
4	Solution Status	Optimal	.
5	Objective Value	12.081337881	12.081338
6			.
7	Primal Infeasibility	1.776357E-15	1.776357E-15
8	Dual Infeasibility	0	0
9	Bound Infeasibility	0	0
10			.
11	Iterations	8	8.000000
12	Presolve Time	0.00	0
13	Solution Time	0.00	0

Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status	Reduced Cost
1	diet		br	N	2.0	0.0	1.7977E308	0.00000	L	1.19066
2	diet		mi	D	3.5	0.0	1	0.05360	B	0.00000
3	diet		ch	N	8.0	0.0	1.7977E308	0.44950	B	0.00000
4	diet		po	N	1.5	0.0	1.7977E308	1.86517	B	0.00000
5	diet		fi	O	11.0	0.5	1.7977E308	0.50000	L	5.15641
6	diet		yo	N	1.0	0.0	1.7977E308	0.00000	L	1.10849

The cost of the optimal diet is 12.08 units.

Example 12.4: Reoptimizing after Modifying the Objective Function

Using the diet problem described in Example 12.3, this example illustrates how to reoptimize an LP problem after modifying the objective function.

Assume that the optimal solution of the diet problem is found and the optimal solutions are stored in the data sets ex3pout and ex3dout.

Suppose the cost of cheese increases from 8 to 10 per unit and the cost of fish decreases from 11 to 7 per serving unit. The COLUMNS section in the input data set ex3 is updated (and the data set is saved as ex4) as follows:

```
COLUMNS      .          .         .     .          .
   ...
.            ch         diet      10    calories   106
   ...
.            fi         diet      7     calories   130
   ...
RHS          .          .         .     .          .
   ...

ENDATA
;
```

You can use the following DATA step to create the data set ex4:

```
data ex4;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME         .          EX4       .     .          .
ROWS         .          .         .     .          .
 N           diet       .         .     .          .
 G           calories   .         .     .          .
 L           protein    .         .     .          .
 G           fat        .         .     .          .
 G           carbs      .         .     .          .
COLUMNS      .          .         .     .          .
.            br         diet      2     calories   90
.            br         protein   4     fat        1
.            br         carbs     15    .          .
.            mi         diet      3.5   calories   120
.            mi         protein   8     fat        5
.            mi         carbs     11.7  .          .
.            ch         diet      10    calories   106
.            ch         protein   7     fat        9
.            ch         carbs     .4    .          .
.            po         diet      1.5   calories   97
.            po         protein   1.3   fat        .1
.            po         carbs     22.6  .          .
.            fi         diet       7    calories   130
.            fi         protein   8     fat        7
.            fi         carbs     0     .          .
.            yo         diet      1     calories   180
.            yo         protein   9.2   fat        1
.            yo         carbs     17    .          .
RHS          .          .         .     .          .
.            .          calories  300   protein    10
.            .          fat       8     carbs      10
BOUNDS       .          .         .     .          .
UP           .          mi        1     .          .
LO           .          fi        .5    .          .
ENDATA       .          .         .     .          .
;
```

You can use the BASIS=WARMSTART option (and the ex3pout and ex3dout data sets from Example 12.3) in the following call to PROC OPTLP to solve the modified problem:

```
proc optlp data=ex4
   presolver = none
   basis     = warmstart
   primalin  = ex3pout
   dualin    = ex3dout
   algorithm = primal
   primalout = ex4pout
   dualout   = ex4dout
   logfreq   = 1;
run;
```

The following iteration log indicates that it takes the primal simplex algorithm no extra iterations to solve the modified problem by using BASIS=WARMSTART, since the optimal solution to the LP problem in Example 12.3 remains optimal after the objective function is changed.

Output 12.4.1 Iteration Log

```
NOTE: The problem EX4 has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.
                          Objective                Entering      Leaving
      Phase Iteration        Value         Time     Variable      Variable
       P 2          1    1.098034E+01          0
NOTE: Optimal.
NOTE: Objective = 10.980335514.
NOTE: The Primal Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.EX4POUT has 6 observations and 10 variables.
NOTE: The data set WORK.EX4DOUT has 4 observations and 10 variables.
```

Note that the primal simplex algorithm is preferred because the primal solution to the original LP is still feasible for the modified problem in this case.

Example 12.5: Reoptimizing after Modifying the Right-Hand Side

You can also modify the right-hand side of your problem and use the BASIS=WARMSTART option to obtain an optimal solution more quickly. Since the dual solution to the original LP is still feasible for the modified problem in this case, the dual simplex algorithm is preferred. This case is illustrated by using the same diet problem as in Example 12.3. Assume that you now need a diet that supplies at least 150 calories. The RHS section in the input data set ex3 is updated (and the data set is saved as ex5) as follows:

```
   ...
RHS          .          .          .     .          .
.            .          calories   150   protein    10
.            .          fat        8     carbs      10
BOUNDS       .          .          .     .          .
   ...
```

You can use the following DATA step to create the data set ex5:

```
data ex5;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME         .          EX5        .     .          .
ROWS         .          .          .     .          .
 N           diet       .          .     .          .
 G           calories   .          .     .          .
 L           protein    .          .     .          .
 G           fat        .          .     .          .
 G           carbs      .          .     .          .
COLUMNS      .          .          .     .          .
.            br         diet       2     calories   90
.            br         protein    4     fat        1
.            br         carbs      15    .          .
.            mi         diet       3.5   calories   120
.            mi         protein    8     fat        5
.            mi         carbs      11.7  .          .
.            ch         diet       8     calories   106
.            ch         protein    7     fat        9
.            ch         carbs      .4    .          .
.            po         diet       1.5   calories   97
.            po         protein    1.3   fat        .1
.            po         carbs      22.6  .          .
.            fi         diet       11    calories   130
.            fi         protein    8     fat        7
.            fi         carbs      0     .          .
.            yo         diet       1     calories   180
.            yo         protein    9.2   fat        1
.            yo         carbs      17    .          .
RHS          .          .          .     .          .
.            .          calories   150   protein    10
.            .          fat        8     carbs      10
BOUNDS       .          .          .     .          .
UP           .          mi         1     .          .
LO           .          fi         .5    .          .
ENDATA       .          .          .     .          .
;
```

You can use the BASIS=WARMSTART option in the following call to PROC OPTLP to solve the modified problem:

```
proc optlp data=ex5
   presolver = none
   basis     = warmstart
   primalin  = ex3pout
   dualin    = ex3dout
   algorithm = dual
   primalout = ex5pout
   dualout   = ex5dout
   logfreq   = 1;
run;
```

Note that the dual simplex algorithm is preferred because the dual solution to the last solved LP is still feasible for the modified problem in this case.

The following iteration log indicates that it takes the dual simplex algorithm just one more phase II iteration to solve the modified problem by using BASIS=WARMSTART.

Output 12.5.1 Iteration Log

```
NOTE: The problem EX5 has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective                Entering      Leaving
      Phase Iteration        Value         Time     Variable      Variable
       D 2          1    8.813205E+00          0   calories (S)      carbs (S)
       D 2          2    9.174413E+00          0
NOTE: Optimal.
NOTE: Objective = 9.1744131985.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.EX5POUT has 6 observations and 10 variables.
NOTE: The data set WORK.EX5DOUT has 4 observations and 10 variables.
```

Compare this with the following call to PROC OPTLP:

```
proc optlp data=ex5
   presolver = none
   algorithm = dual
   logfreq   = 1;
run;
```

This call to PROC OPTLP solves the modified problem "from scratch" (without using the BASIS=WARMSTART option) and produces the following iteration log.

Output 12.5.2 Iteration Log

```
NOTE: The problem EX5 has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective                Entering      Leaving
      Phase Iteration       Value         Time     Variable      Variable
       D 1          1    0.000000E+00          0
       D 2          2    5.500000E+00          0         mi           fat (S)
       D 2          3    8.650000E+00          0         ch       protein (S)
       D 2          4    8.925676E+00          0         po         carbs (S)
       D 2          5    9.174413E+00          0
NOTE: Optimal.
NOTE: Objective = 9.1744131985.
NOTE: The Dual Simplex solve time is 0.00 seconds.
```

It is clear that using the BASIS=WARMSTART option saves computation time. For larger or more complex examples, the benefits of using this option are more pronounced.

Example 12.6: Reoptimizing after Adding a New Constraint

Assume that after solving the diet problem in Example 12.3 you need to add a new constraint on sodium intake of no more than 550 mg/day for adults. The updated nutrition data are given in Table 12.15.

Table 12.15 Updated Cost and Nutrition Values

	Bread	Milk	Cheese	Potato	Fish	Yogurt
Cost	2.0	3.5	8.0	1.5	11.0	1.0
Protein, g	4.0	8.0	7.0	1.3	8.0	9.2
Fat, g	1.0	5.0	9.0	0.1	7.0	1.0
Carbohydrates, g	15.0	11.7	0.4	22.6	0.0	17.0
Calories, Cal	90	120	106	97	130	180
sodium, mg	148	122	337	186	56	132

The input data set ex3 is updated (and the data set is saved as ex6) as follows:

```
/* added a new constraint to the diet problem */
data ex6;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME          .           EX6       .     .          .
ROWS          .           .         .     .          .
 N            diet        .         .     .          .
 G            calories    .         .     .          .
 L            protein     .         .     .          .
```

```
 G          fat        .          .      .          .
 G          carbs      .          .      .          .
 L          sodium     .          .      .          .
COLUMNS     .          .          .      .          .
.           br         diet       2      calories   90
.           br         protein    4      fat        1
.           br         carbs      15     sodium     148
.           mi         diet       3.5    calories   120
.           mi         protein    8      fat        5
.           mi         carbs      11.7   sodium     122
.           ch         diet       8      calories   106
.           ch         protein    7      fat        9
.           ch         carbs      .4     sodium     337
.           po         diet       1.5    calories   97
.           po         protein    1.3    fat        .1
.           po         carbs      22.6   sodium     186
.           fi         diet       11     calories   130
.           fi         protein    8      fat        7
.           fi         carbs      0      sodium     56
.           yo         diet       1      calories   180
.           yo         protein    9.2    fat        1
.           yo         carbs      17     sodium     132
RHS         .          .          .      .          .
.           .          calories   300    protein    10
.           .          fat        8      carbs      10
.           .          sodium     550    .          .
BOUNDS      .          .          .      .          .
UP          .          mi         1      .          .
LO          .          fi         .5     .          .
ENDATA      .          .          .      .          .
;
```

For the modified problem you can warm start the primal and dual simplex algorithms to get a solution faster. The dual simplex algorithm is preferred because a dual feasible solution can be readily constructed from the optimal solution to the diet optimization problem.

Since there is a new constraint in the modified problem, you can use the following SAS code to create a new DUALIN= data set ex6din with this information:

```
data ex6newcon;
   _ROW_='sodium  '; _STATUS_='A';
   output;
run;

/* create a new DUALIN= data set to include the new constraint */
data ex6din;
   set ex3dout ex6newcon;
run;
```

Note that this step is optional. In this example, you can still use the data set ex3dout as the DUALIN= data set to solve the modified LP problem by using the BASIS=WARMSTART option. PROC OPTLP validates the PRIMALIN= and DUALIN= data sets against the input model. Any new variable (or constraint) in the model is added to the PRIMALIN= (or DUALIN=) data set, and its status is assigned to be 'A'. The primal and dual simplex algorithms decide its corresponding status internally. Any variable in the PRIMALIN= and DUALIN= data sets but not in the input model is removed.

The _ROW_ and _STATUS_ columns of the DUALIN= data set ex6din are shown in Output 12.6.1.

Output 12.6.1 DUALIN= Data Set with a Newly Added Constraint

Obs	_ROW_	_STATUS_
1	calories	U
2	protein	L
3	fat	U
4	carbs	B
5	sodium	A

The dual simplex algorithm is called to solve the modified diet optimization problem more quickly with the following SAS code:

```
proc optlp data=ex6
   objsense=min
   presolver=none
   algorithm=ds
   primalout=ex6pout
   dualout=ex6dout
   scale=none
   logfreq=1
   basis=warmstart
   primalin=ex3pout
   dualin=ex6din;
run;
```

The optimal primal and dual solutions of the modified problem are displayed in Output 12.6.2.

Output 12.6.2 Primal and Dual Solution Output

Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status	Reduced Cost
1	diet		br	N	2.0	0.0	1.7977E308	0.00000	L	1.19066
2	diet		mi	D	3.5	0.0	1	0.05360	B	0.00000
3	diet		ch	N	8.0	0.0	1.7977E308	0.44950	B	0.00000
4	diet		po	N	1.5	0.0	1.7977E308	1.86517	B	0.00000
5	diet		fi	O	11.0	0.5	1.7977E308	0.50000	L	5.15641
6	diet		yo	N	1.0	0.0	1.7977E308	0.00000	L	1.10849

Dual Solution

Obs	Objective Function ID	RHS ID	Constraint Name	Constraint Type	Constraint RHS	Constraint Lower Bound	Constraint Upper Bound	Dual Variable Value	Constraint Status	Constraint Activity
1	diet		calories	G	300	.	.	0.02179	U	300.000
2	diet		protein	L	10	.	.	-0.55360	L	10.000
3	diet		fat	G	8	.	.	1.06286	U	8.000
4	diet		carbs	G	10	.	.	0.00000	B	42.960
5	diet		sodium	L	550	.	.	0.00000	B	532.941

The iteration log shown in Output 12.6.3 indicates that it takes the dual simplex algorithm no more iterations to solve the modified problem by using the BASIS=WARMSTART option, since the optimal solution to the original problem remains optimal after one more constraint is added.

Output 12.6.3 Iteration Log

```
NOTE: The problem EX6 has 6 variables (0 free, 0 fixed).
NOTE: The problem has 5 constraints (2 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 29 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective                Entering      Leaving
      Phase Iteration       Value         Time     Variable      Variable
       D 2          1    1.208134E+01         0
NOTE: Optimal.
NOTE: Objective = 12.081337881.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The data set WORK.EX6POUT has 6 observations and 10 variables.
NOTE: The data set WORK.EX6DOUT has 5 observations and 10 variables.
```

Both this example and Example 12.4 illustrate the situation in which the optimal solution does not change after some perturbation of the parameters of the LP problem. The simplex algorithm starts from an optimal solution and quickly verifies the optimality. Usually the optimal solution of the slightly perturbed problem can be obtained after performing relatively small number of iterations if starting with the optimal solution of the original problem. In such cases you can expect a dramatic reduction of computation time, for instance, if you want to solve a large LP problem and a slightly perturbed version of this problem by using the BASIS=WARMSTART option rather than solving both problems from scratch.

Example 12.7: Finding an Irreducible Infeasible Set

This example demonstrates the use of the IIS= option to locate an irreducible infeasible set. Suppose you want to solve a linear program that has the following simple formulation:

$$
\begin{array}{rrcrcrcrl}
\min & & x_1 & + & x_2 & + & x_3 & & \text{(cost)} \\
\text{subject to} & & x_1 & + & x_2 & & & \geq \quad 10 & \text{(con1)} \\
 & & x_1 & & & + & x_3 & \leq \quad 4 & \text{(con2)} \\
 & 4 \leq & & & x_2 & + & x_3 & \leq \quad 5 & \text{(con3)} \\
 & & & & & & x_1, \; x_2 & \geq \quad 0 & \\
 & & & & 0 & \leq & x_3 & \leq \quad 3 &
\end{array}
$$

The corresponding MPS-format SAS data set is as follows:

```
/* infeasible */
data exiis;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
datalines;
NAME      .        .        .        .        .
ROWS      .        .        .        .        .
 N        cost     .        .        .        .
 G        con1     .        .        .        .
 L        con2     .        .        .        .
 G        con3     .        .        .        .
COLUMNS   .        .        .        .        .
.         x1       cost     1        con1     1
.         x1       con2     1        .        .
.         x2       cost     1        con1     1
.         x2       con3     1        .        .
.         x3       cost     1        con2     1
.         x3       con3     1        .        .
RHS       .        .        .        .        .
.         rhs      con1     10       con2     4
.         rhs      con3     4        .        .
RANGES    .        .        .        .        .
.         r1       con3     1        .        .
BOUNDS    .        .        .        .        .
UP        b1       x3       3        .        .
ENDATA    .        .        .        .        .
;
```

It is easy to verify that the following three constraints (or rows) and one variable (or column) bound form an IIS for this problem.

$$
\begin{array}{rcrcrcll}
x_1 & + & x_2 & & & \geq & 10 & \text{(con1)} \\
x_1 & & & + & x_3 & \leq & 4 & \text{(con2)} \\
 & & x_2 & + & x_3 & \leq & 5 & \text{(con3)} \\
 & & & & x_3 & \geq & 0 &
\end{array}
$$

You can use the IIS=ON option to detect this IIS by using the following statements:

```
proc optlp data=exiis
   iis=on
   primalout=iis_vars
   dualout=iis_cons
   logfreq=1;
run;
```

The OPTLP procedure outputs the detected IIS to the data sets specified by the PRIMALOUT= and DUALOUT= options, then stops. The notes shown in Output 12.7.1 are printed to the log.

Output 12.7.1 The IIS= Option: Log

```
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 constraints (1 LE, 0 EQ, 1 GE, 1 range).
NOTE: The problem has 6 constraint coefficients.
NOTE: The IIS= option is enabled.
                          Objective                Entering      Leaving
      Phase Iteration       Value         Time     Variable      Variable
       P 1          1    1.400000E+01         0         x2           con3 (S)
       P 1          2    5.000000E+00         0         x1           con2 (S)
       P 1          3    1.000000E+00         0
NOTE: Applying the IIS sensitivity filter.
NOTE: The sensitivity filter removed 1 constraints and 3 variable bounds.
NOTE: Applying the IIS deletion filter.
NOTE: Processing constraints.
      Processed     Removed      Time
              0           0         0
              1           0         0
              2           0         0
              3           0         0
NOTE: Processing variable bounds.
      Processed     Removed      Time
              0           0         0
              1           0         0
              2           0         0
              3           0         0
NOTE: The deletion filter removed 0 constraints and 0 variable bounds.
NOTE: The IIS= option found this problem to be infeasible.
NOTE: The IIS= option found an irreducible infeasible set with 1 variables and
      3 constraints.
NOTE: The IIS solve time is 0.00 seconds.
NOTE: The data set WORK.IIS_VARS has 3 observations and 10 variables.
NOTE: The data set WORK.IIS_CONS has 3 observations and 10 variables.
```

The data sets iis_cons and iis_vars are shown in Output 12.7.2.

Output 12.7.2 Identify Rows and Columns in the IIS

Constraints in the IIS

Obs	Objective Function ID	RHS ID	Constraint Name	Constraint Type	Constraint RHS	Constraint Lower Bound	Constraint Upper Bound	Dual Variable Value	Constraint Status	Constraint Activity
1	cost	rhs	con1	G	10	.	.	.	I_L	.
2	cost	rhs	con2	L	4	.	.	.	I_U	.
3	cost	rhs	con3	R	.	4	5	.	I_U	.

Variables in the IIS

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status	Reduced Cost
1	cost	rhs	x1	N	1	0	1.7977E308	.		.
2	cost	rhs	x2	N	1	0	1.7977E308	.		.
3	cost	rhs	x3	D	1	0	3	.	I_L	.

The constraint $x_2 + x_3 \leq 5$, which is an element of the IIS, is created by the RANGES section. The original constraint is con3, a "$\geq$" constraint with an RHS value of 4. If you choose to remove the constraint $x_2 + x_3 \leq 5$, you can accomplish this by removing con3 from the RANGES section in the MPS-format SAS data set exiis. Since con3 is the only observation in the section, the identifier observation can also be removed. The modified LP problem is specified in the following SAS statements:

```
 /* dropping con3, feasible */
data exiisf;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
datalines;
NAME      .         .         .         .
ROWS      .         .         .         .
 N        cost      .         .         .
 G        con1      .         .         .
 L        con2      .         .         .
 G        con3      .         .         .
COLUMNS   .         .         .         .
.         x1        cost      1         con1      1
.         x1        con2      1         .         .
.         x2        cost      1         con1      1
.         x2        con3      1         .         .
.         x3        cost      1         con2      1
.         x3        con3      1         .         .
RHS       .         .         .         .         .
.         rhs       con1      10        con2      4
.         rhs       con3      4         .         .
BOUNDS    .         .         .         .         .
UP        b1        x3        3         .         .
ENDATA    .         .         .         .         .
;
```

Since one element of the IIS has been removed, the modified LP problem should no longer contain the infeasible set. Due to the size of this problem, there should be no additional irreducible infeasible sets. You can confirm this by submitting the following SAS statements:

```
proc optlp data=exiisf
   iis=on;
run;
```

The notes shown in Output 12.7.3 are printed to the log.

Output 12.7.3 The IIS= Option: Log

```
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 constraints (1 LE, 0 EQ, 2 GE, 0 range).
NOTE: The problem has 6 constraint coefficients.
NOTE: The IIS= option is enabled.
                          Objective
      Phase Iteration        Value         Time
       P 1          1    1.400000E+01         0
       P 1          3    0.000000E+00         0
NOTE: The IIS= option found this problem to be feasible.
NOTE: The IIS solve time is 0.00 seconds.
NOTE: The data set WORK.EXSS has 8 observations and 3 variables.
```

The solution summary is displayed in Output 12.7.4.

Output 12.7.4 Infeasibility Removed

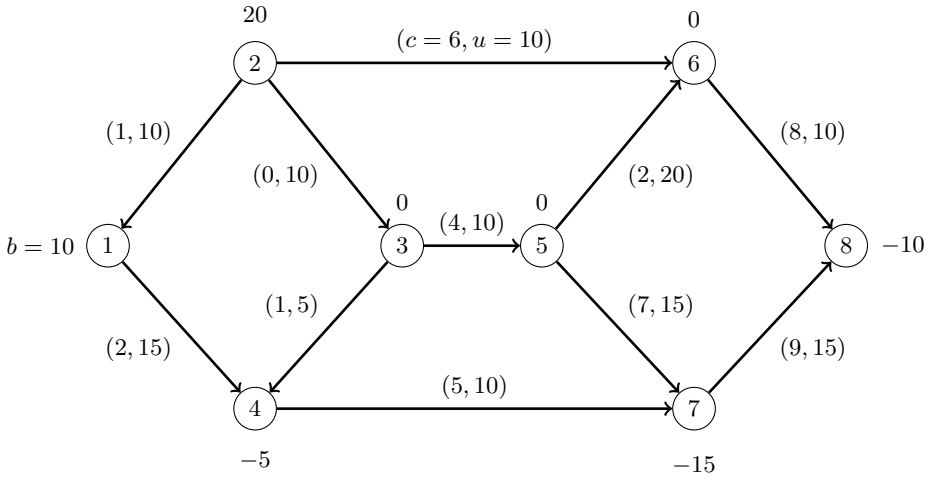
Solution Summary

Obs	Label1	cValue1	nValue1
1	Solver	LP	.
2	Algorithm	IIS	.
3	Objective Function	cost	.
4	Solution Status	Feasible	.
5			.
6	Iterations	3	3.000000
7	Presolve Time	0.00	0
8	Solution Time	0.00	0

Example 12.8: Using the Network Simplex Algorithm

This example demonstrates how to use the network simplex algorithm to find the minimum-cost flow in a directed graph. Consider the directed graph in Figure 12.5, which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 12.5 Minimum-Cost Network Flow Problem: Data



You can use the following SAS statements to create the input data set ex8:

```
data ex8;
   input field1 $8. field2 $13. @25 field3 $13. field4  @53 field5 $13. field6;
   datalines;
NAME     .              .                .      .                  .
ROWS     .              .                .      .                  .
N        obj            .                .      .                  .
E        balance['1']   .                .      .                  .
E        balance['2']   .                .      .                  .
E        balance['3']   .                .      .                  .
E        balance['4']   .                .      .                  .
E        balance['5']   .                .      .                  .
E        balance['6']   .                .      .                  .
E        balance['7']   .                .      .                  .
E        balance['8']   .                .      .                  .
COLUMNS  .              .                .      .                  .
.        x['1','4']     obj              2      balance['1']       1
.        x['1','4']     balance['4']    -1      .                  .
.        x['2','1']     obj              1      balance['1']      -1
.        x['2','1']     balance['2']     1      .                  .
.        x['2','3']     balance['2']     1      balance['3']      -1
.        x['2','6']     obj              6      balance['2']       1
.        x['2','6']     balance['6']    -1      .                  .
.        x['3','4']     obj              1      balance['3']       1
.        x['3','4']     balance['4']    -1      .                  .
.        x['3','5']     obj              4      balance['3']       1
.        x['3','5']     balance['5']    -1      .                  .
.        x['4','7']     obj              5      balance['4']       1
```

```
.       x['4','7']      balance['7']       -1      .                .
.       x['5','6']      obj                 2      balance['5']     1
.       x['5','6']      balance['6']       -1      .                .
.       x['5','7']      obj                 7      balance['5']     1
.       x['5','7']      balance['7']       -1      .                .
.       x['6','8']      obj                 8      balance['6']     1
.       x['6','8']      balance['8']       -1      .                .
.       x['7','8']      obj                 9      balance['7']     1
.       x['7','8']      balance['8']       -1      .                .
RHS     .               .                   .      .                .
.       .RHS.           balance['1']       10      .                .
.       .RHS.           balance['2']       20      .                .
.       .RHS.           balance['4']       -5      .                .
.       .RHS.           balance['7']      -15      .                .
.       .RHS.           balance['8']      -10      .                .
BOUNDS  .               .                   .      .                .
UP      .BOUNDS.        x['1','4']         15      .                .
UP      .BOUNDS.        x['2','1']         10      .                .
UP      .BOUNDS.        x['2','3']         10      .                .
UP      .BOUNDS.        x['2','6']         10      .                .
UP      .BOUNDS.        x['3','4']          5      .                .
UP      .BOUNDS.        x['3','5']         10      .                .
UP      .BOUNDS.        x['4','7']         10      .                .
UP      .BOUNDS.        x['5','6']         20      .                .
UP      .BOUNDS.        x['5','7']         15      .                .
UP      .BOUNDS.        x['6','8']         10      .                .
UP      .BOUNDS.        x['7','8']         15      .                .
ENDATA  .               .                   .      .                .
;
```

You can use the following call to PROC OPTLP to find the minimum-cost flow:

```
proc optlp
  presolver  = none
  printlevel = 2
  logfreq    = 1
  data       = ex8
  primalout  = ex8out
  algorithm  = ns;
run;
```

The optimal solution is displayed in Output 12.8.1.

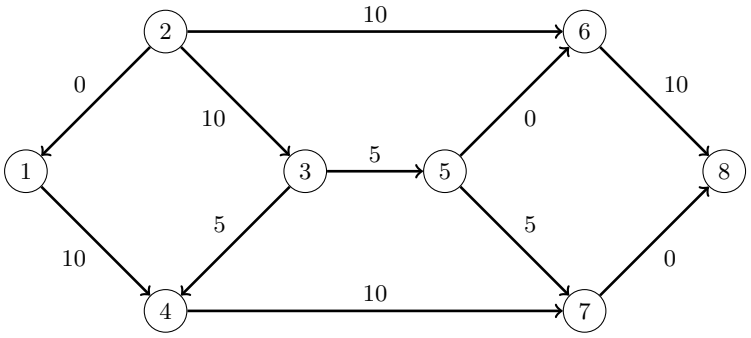
Output 12.8.1 Network Simplex Algorithm: Primal Solution Output

Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status	Reduced Cost
1	obj	.RHS.	x['1','4']	D	2	0	15	10	B	0
2	obj	.RHS.	x['2','1']	D	1	0	10	0	L	2
3	obj	.RHS.	x['2','3']	D	0	0	10	10	B	0
4	obj	.RHS.	x['2','6']	D	6	0	10	10	B	0
5	obj	.RHS.	x['3','4']	D	1	0	5	5	B	0
6	obj	.RHS.	x['3','5']	D	4	0	10	5	B	0
7	obj	.RHS.	x['4','7']	D	5	0	10	10	U	-5
8	obj	.RHS.	x['5','6']	D	2	0	20	0	L	0
9	obj	.RHS.	x['5','7']	D	7	0	15	5	B	0
10	obj	.RHS.	x['6','8']	D	8	0	10	10	B	0
11	obj	.RHS.	x['7','8']	D	9	0	15	0	L	6

The optimal solution is represented graphically in Figure 12.6.

Figure 12.6 Minimum-Cost Network Flow Problem: Optimal Solution



The iteration log is displayed in Output 12.8.2.

Output 12.8.2 Log: Solution Progress

```
NOTE: The problem has 11 variables (0 free, 0 fixed).
NOTE: The problem has 8 constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 22 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Network Simplex algorithm is used.
NOTE: The network has 8 rows (100.00%), 11 columns (100.00%), and 1 component.
NOTE: The network extraction and setup time is 0.00 seconds.
                      Primal         Primal           Dual
     Iteration      Objective  Infeasibility  Infeasibility     Time
             1   0.000000E+00   2.000000E+01   8.900000E+01     0.00
             2   0.000000E+00   2.000000E+01   8.900000E+01     0.00
             3   5.000000E+00   1.500000E+01   8.400000E+01     0.00
             4   5.000000E+00   1.500000E+01   8.300000E+01     0.00
             5   7.500000E+01   1.500000E+01   8.300000E+01     0.00
             6   7.500000E+01   1.500000E+01   7.900000E+01     0.00
             7   1.300000E+02   1.000000E+01   7.600000E+01     0.00
             8   2.700000E+02   0.000000E+00   0.000000E+00     0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.00 seconds.
NOTE: Optimal.
NOTE: Objective = 270.
NOTE: The data set WORK.EX8OUT has 11 observations and 10 variables.
```

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice-Hall.

Andersen, E. D., and Andersen, K. D. (1995). "Presolving in Linear Programming." *Mathematical Programming* 71:221–245.

Chinneck, J. W. (2008). *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods.* Vol. 118 of International Series in Operations Research and Management Science. New York: Springer.

Dantzig, G. B. (1963). *Linear Programming and Extensions*. Princeton, NJ: Princeton University Press.

Forrest, J. J., and Goldfarb, D. (1992). "Steepest-Edge Simplex Algorithms for Linear Programming." *Mathematical Programming* 5:1–28.

Gondzio, J. (1997). "Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method." *INFORMS Journal on Computing* 9:73–91.

Harris, P. M. J. (1973). "Pivot Selection Methods in the Devex LP Code." *Mathematical Programming* 57:341–374.

Maros, I. (2003). *Computational Techniques of the Simplex Method.* Boston: Kluwer Academic.

Chapter 13
The OPTMILP Procedure

Contents

Overview: OPTMILP Procedure **625**
Getting Started: OPTMILP Procedure **626**
Syntax: OPTMILP Procedure **629**
- Functional Summary 629
- PROC OPTMILP Statement 630
- Decomposition Algorithm Statements 640
- PERFORMANCE Statement 640
- TUNER Statement 640

Details: OPTMILP Procedure **641**
- Data Input and Output 641
- Warm Start 643
- Branch-and-Bound Algorithm 643
- Controlling the Branch-and-Bound Algorithm 644
- Presolve and Probing 646
- Cutting Planes 646
- Primal Heuristics 648
- Parallel Processing 649
- Node Log 649
- ODS Tables 650
- Macro Variable _OROPTMILP_ 654

Examples: OPTMILP Procedure **657**
- Example 13.1: Simple Integer Linear Program 657
- Example 13.2: MIPLIB Benchmark Instance 662
- Example 13.3: Facility Location 666
- Example 13.4: Scheduling 674

References **684**

Overview: OPTMILP Procedure

The OPTMILP procedure solves general mixed integer linear programs (MILPs).

A standard mixed integer linear program has the formulation

$$\begin{array}{rl} \min & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \\ & \mathbf{x}_i \in \mathbb{Z} \quad \forall i \in \mathcal{S} \end{array} \qquad \text{(MILP)}$$

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of structural variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m\times n}$	is the matrix of technological coefficients
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^n$	is the vector of lower bounds on variables
$\mathbf{u}$	$\in$	$\mathbb{R}^n$	is the vector of upper bounds on variables
$\mathcal{S}$			is a nonempty subset of the set $\{1 \ldots, n\}$ of indices

The OPTMILP procedure implements a linear-programming-based branch-and-cut algorithm. This divide-and-conquer approach attempts to solve the original problem by solving linear programming relaxations of a sequence of smaller subproblems. The OPTMILP procedure also implements advanced techniques such as presolving, generating cutting planes, and applying primal heuristics to improve the efficiency of the overall algorithm.

The OPTMILP procedure requires a mixed integer linear program to be specified using a SAS data set that adheres to the mathematical programming system (MPS) format, a widely accepted format in the optimization community. Chapter 17 discusses the MPS format in detail. It is also possible to input an incumbent solution in MPS format; see the section "Warm Start" on page 643 for details.

You can use the MPSOUT= option to convert data sets that are formatted for the LP procedure into MPS-format SAS data sets. The option is available in the LP, INTPOINT, and NETFLOW procedures. For details about this option, see Chapter 4, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), Chapter 3, "The INTPOINT Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), and Chapter 5, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

The OPTMILP procedure provides various control options and solution strategies. In particular, you can enable, disable, or set levels for the advanced techniques previously mentioned.

The OPTMILP procedure outputs an optimal solution or the best feasible solution found, if any, in SAS data sets. This enables you to generate solution reports and perform additional analyses by using SAS software.

Getting Started: OPTMILP Procedure

The following example illustrates the use of the OPTMILP procedure to solve mixed integer linear programs. For more examples, see the section "Examples: OPTMILP Procedure" on page 657. Suppose you want to solve the following problem:

$$
\begin{array}{lrcrcrcrl}
\min & 2x_1 & - & 3x_2 & - & 4x_3 & & & \\
\text{s.t.} & & - & 2x_2 & - & 3x_3 & \geq & -5 & \text{(R1)} \\
 & x_1 & + & x_2 & + & 2x_3 & \leq & 4 & \text{(R2)} \\
 & x_1 & + & 2x_2 & + & 3x_3 & \leq & 7 & \text{(R3)} \\
 & & & x_1, & x_2, & x_3 & \geq & 0 & \\
 & & & x_1, & x_2, & x_3 & \in & \mathbb{Z} &
\end{array}
$$

The corresponding MPS-format SAS data set follows:

```
data ex_mip;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME          .       EX_MIP       .   .         .
ROWS          .         .          .   .         .
N             COST      .          .   .         .
G             R1        .          .   .         .
L             R2        .          .   .         .
L             R3        .          .   .         .
COLUMNS       .         .          .   .         .
.             MARK00  'MARKER'     .   'INTORG'  .
.             X1      COST         2   R2        1
.             X1      R3           1   .         .
.             X2      COST        -3   R1       -2
.             X2      R2           1   R3        2
.             X3      COST        -4   R1       -3
.             X3      R2           2   R3        3
.             MARK01  'MARKER'     .   'INTEND'  .
RHS           .       .            .   .         .
.             RHS     R1          -5   R2        4
.             RHS     R3           7   .         .
ENDATA        .       .            .   .         .
;
```

You can also create this SAS data set from an MPS-format flat file (ex_mip.mps) by using the following SAS macro:

```
%mps2sasd(mpsfile = "ex_mip.mps", outdata = ex_mip);
```

This problem can be solved by using the following statement to call the OPTMILP procedure:

```
proc optmilp data = ex_mip
   objsense   = min
   primalout  = primal_out
   dualout    = dual_out
   presolver  = automatic
   heuristics = automatic;
run;
```

The DATA= option names the MPS-format SAS data set that contains the problem data. The OBJSENSE= option specifies whether to maximize or minimize the objective function. The PRIMALOUT= option names the SAS data set to contain the optimal solution or the best feasible solution found by the solver. The

DUALOUT= option names the SAS data set to contain the constraint activities. The PRESOLVER= and HEURISTICS= options specify the levels for presolving and applying heuristics, respectively. In this example, each option is set to its default value AUTOMATIC, meaning that the solver automatically determines the appropriate levels for presolve and heuristics.

The optimal integer solution and its corresponding constraint activities, stored in the data sets primal_out and dual_out, respectively, are displayed in Figure 13.1 and Figure 13.2.

Figure 13.1 Optimal Solution

The OPTMILP Procedure
Primal Integer Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value
1	COST	RHS	X1	B	2	0	1	0
2	COST	RHS	X2	B	-3	0	1	1
3	COST	RHS	X3	B	-4	0	1	1

Figure 13.2 Constraint Activities

The OPTMILP Procedure
Constraint Information

Obs	Objective Function ID	RHS ID	Constraint Name	Constraint Type	Constraint RHS	Constraint Lower Bound	Constraint Upper Bound	Constraint Activity
1	COST	RHS	R1	G	-5	.	.	-5
2	COST	RHS	R2	L	4	.	.	3
3	COST	RHS	R3	L	7	.	.	5

The solution summary stored in the macro variable _OROPTMILP_ can be viewed by issuing the following statement:

```
%put &_OROPTMILP_;
```

This produces the output shown in Figure 13.3.

Figure 13.3 Macro Output

```
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=-7 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0 BEST_BOUND=-7 NODES=0 ITERATIONS=0 PRESOLVE_TIME=0.35
SOLUTION_TIME=0.35
```

See the section "Data Input and Output" on page 641 for details about the type and status codes displayed for variables and constraints.

Syntax: OPTMILP Procedure

The following statements are available in the OPTMILP procedure:

PROC OPTMILP < *options* > ;
  DECOMP < *options* > ;
  DECOMP_MASTER < *options* > ;
  DECOMP_MASTER_IP < *options* > ;
  DECOMP_SUBPROB < *options* > ;
  PERFORMANCE < *performance-options* > ;
  TUNER < *tuner-options* > ;

Functional Summary

Table 13.1 summarizes the options available for the OPTMILP procedure, classified by function.

Table 13.1 Options for the OPTMILP Procedure

Description	Option
Data Set Options	
Specifies the input data set	DATA=
Specifies the constraint activities output data set	DUALOUT=
Specifies whether the MILP model is a maximization or minimization problem	OBJSENSE=
Specifies the primal solution input data set (warm start)	PRIMALIN=
Specifies the primal solution output data set	PRIMALOUT=
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Control Options	
Specifies the stopping criterion based on absolute objective gap	ABSOBJGAP=
Specifies the cutoff value for node removal	CUTOFF=
Emphasizes feasibility or optimality	EMPHASIS=
Specifies the maximum violation on variables and constraints	FEASTOL=
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=
Specifies the frequency of printing the node log	LOGFREQ=
Specifies the detail of solution progress printed in log	LOGLEVEL=
Specifies the maximum number of nodes to be processed	MAXNODES=
Specifies the maximum number of solutions to be found	MAXSOLS=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the tolerance used in determining the optimality of nodes in the branch-and-bound tree	OPTTOL=
Toggles ODS output	PRINTLEVEL=
Specifies the probing level	PROBE=
Specifies the stopping criterion based on relative objective gap	RELOBJGAP=
Specifies the scale of the problem matrix	SCALE=
Specifies the initial seed for the random number generator	SEED=

Table 13.1 (continued)

Description	Option
Specifies the stopping criterion based on target objective value	TARGET=
Specifies whether time units are CPU time or real time	TIMETYPE=
Heuristics Option	
Specifies the primal heuristics level	HEURISTICS=
Search Options	
Specifies the level of conflict search	CONFLICTSEARCH=
Specifies the node selection strategy	NODESEL=
Enables use of variable priorities	PRIORITY=
Specifies the restarting strategy	RESTARTS=
Specifies the number of simplex iterations performed on each variable in strong branching strategy	STRONGITER=
Specifies the number of candidates for strong branching	STRONGLEN=
Specifies the level of symmetry detection	SYMMETRY=
Specifies the rule for selecting branching variable	VARSEL=
Cut Options	
Specifies the cut level for all cuts	ALLCUTS=
Specifies the clique cut level	CUTCLIQUE=
Specifies the flow cover cut level	CUTFLOWCOVER=
Specifies the flow path cut level	CUTFLOWPATH=
Specifies the Gomory cut level	CUTGOMORY=
Specifies the generalized upper bound (GUB) cover cut level	CUTGUB=
Specifies the implied bounds cut level	CUTIMPLIED=
Specifies the knapsack cover cut level	CUTKNAPSACK=
Specifies the lift-and-project cut level	CUTLAP=
Specifies the mixed lifted 0-1 cut level	CUTMILIFTED=
Specifies the mixed integer rounding (MIR) cut level	CUTMIR=
Specifies the multicommodity network flow cut level	CUTMULTICOMMODITY=
Specifies the row multiplier factor for cuts	CUTSFACTOR=
Specifies the overall cut aggressiveness	CUTSTRATEGY=
Specifies the zero-half cut level	CUTZEROHALF=

PROC OPTMILP Statement

PROC OPTMILP < *options* > **;**

You can specify the following options in the PROC OPTMILP statement.

Data Set Options

DATA=*SAS-data-set*

specifies the input data set that corresponds to the MILP model. If this option is not specified, PROC OPTMILP uses the most recently created SAS data set. See Chapter 17, "The MPS-Format SAS Data Set," for more details about the input data set.

DUALOUT=*SAS-data-set*

DOUT=*SAS-data-set*

specifies the output data set to contain the constraint activities.

OBJSENSE=MIN | MAX

specifies whether the MILP model is a minimization or a maximization problem. You can use OBJSENSE=MIN for a minimization problem and OBJSENSE=MAX for a maximization problem. Alternatively, you can specify the objective sense in the input data set. This option supersedes the objective sense specified in the input data set. If the objective sense is not specified anywhere, then PROC OPTMILP interprets and solves the MILP as a minimization problem.

PRIMALIN=*SAS-data-set*

enables you to input a warm start solution in a SAS data set. PROC OPTMILP validates both the data set and the solution stored in the data set. If the data set is not valid, then the PRIMALIN= data are ignored. If the solution stored in a valid PRIMALIN= data set is a feasible integer solution, then it provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution stored in a valid PRIMALIN= data set is infeasible, contains missing values, or contains fractional values for integer variables, PROC OPTMILP tries to repair the solution with a number of specialized repair heuristics. See the section "Warm Start" on page 643 for details.

PRIMALOUT=*SAS-data-set*

POUT=*SAS-data-set*

specifies the output data set for the primal solution. This data set contains the primal solution information. See the section "Data Input and Output" on page 641 for details.

Presolve Option

PRESOLVER=*number* | *string*

specifies a presolve *string* or its corresponding value *number*, as listed in Table 13.2.

Table 13.2 Values for PRESOLVER= Option

number	*string*	**Description**
–1	AUTOMATIC	Applies the default level of presolve processing
0	NONE	Disables presolver
1	BASIC	Performs minimal presolve processing
2	MODERATE	Applies a higher level of presolve processing
3	AGGRESSIVE	Applies the highest level of presolve processing

The default value is AUTOMATIC.

Control Options

ABSOBJGAP=*number*

specifies a stopping criterion. When the absolute difference between the best integer objective and the best bound on the objective function value becomes smaller than the value of *number*, the procedure stops. The value of *number* can be any nonnegative number; the default value is 1E–6.

CUTOFF=*number*

cuts off any nodes in a minimization (maximization) problem that have an objective value at or above (below) *number*. The value of *number* can be any number; the default value is the positive (negative) number that has the largest absolute value that can be represented in your operating environment.

EMPHASIS=*number* | *string*

specifies a search emphasis *string* or its corresponding value *number* as listed in Table 13.3.

Table 13.3 Values for EMPHASIS= Option

number	*string*	**Description**
0	BALANCE	Performs a balanced search
1	OPTIMAL	Emphasizes optimality over feasibility
2	FEASIBLE	Emphasizes feasibility over optimality

The default value is BALANCE.

FEASTOL=*number*

specifies the tolerance that PROC OPTMILP uses to check the feasibility of a solution. This tolerance applies both to the maximum violation of bounds on variables and to the difference between the right-hand sides and left-hand sides of constraints. The value of *number* can be any value between 1E–4 and 1E–9, inclusive. However, the value of *number* cannot be larger than the integer feasibility tolerance. If the value of *number* is larger than the value of the INTTOL= option, then PROC OPTMILP sets FEASTOL= to the value of INTTOL=. The default value is 1E–6.

If PROC OPTMILP fails to find a feasible solution within this tolerance but does find a solution that has some violation, then the procedure stops with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMILP_" on page 654).

INTTOL=*number*

specifies the amount by which an integer variable value can differ from an integer and still be considered integer feasible. The value of *number* can be any number between 1E–9 and 0.5, inclusive. PROC OPTMILP attempts to find an optimal solution whose integer infeasibility is less than *number*. The default value is 1E–5.

If the best solution that PROC OPTMILP finds has an integer infeasibility larger than the value of *number*, then PROC OPTMILP stops with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMILP_" on page 654).

LOGFREQ=*number*

PRINTFREQ=*number*

specifies how often information is printed in the node log. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is 100. If *number* is set to 0, then the node log is disabled. If *number* is positive, then an entry is made in the node log at the first node, at the last node, and at intervals dictated by the value of *number*. An entry is also made each time a better integer solution is found.

LOGLEVEL=*number* | *string*

PRINTLEVEL2=*number* | *string*

controls the amount of information displayed in the SAS log by the solver, from a short description of presolve information and summary to details at each node. Table 13.4 describes the valid values for this option.

Table 13.4 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all solver-related messages in the SAS log
1	BASIC	Displays a solver summary after stopping
2	MODERATE	Prints a solver summary and a node log by using the interval dictated by the LOGFREQ= option
3	AGGRESSIVE	Prints a detailed solver summary and a node log by using the interval dictated by the LOGFREQ= option

The default value is MODERATE.

MAXNODES=*number*

specifies the maximum number of branch-and-bound nodes to be processed. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is $2^{31} - 1$.

MAXSOLS=*number*

specifies a stopping criterion. If *number* solutions have been found, then the procedure stops. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is $2^{31} - 1$.

MAXTIME=*t*

specifies an upper limit of *t* seconds of time for reading in the data and performing the optimization process. The value of the TIMETYPE= option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OPTTOL=*number*

specifies the tolerance used to determine the optimality of nodes in the branch-and-bound tree. The value of *number* can be any value between (and including) 1E–4 and 1E–9. The default value is 1E–6.

PRINTLEVEL=0 | **1** | **2**

specifies whether a summary of the problem and solution should be printed. If PRINTLEVEL=1, then the Output Delivery System (ODS) tables ProblemSummary, SolutionSummary, and PerformanceInfo are produced and printed. If PRINTLEVEL=2, then the same tables are produced and printed along with an additional table called ProblemStatistics. If PRINTLEVEL=0, then no ODS tables are produced or printed. The default value is 1.

For details about the ODS tables created by PROC OPTMILP, see the section "ODS Tables" on page 650.

PROBE=*number* | *string*
specifies a probing *string* or its corresponding value *number*, as listed in Table 13.5.

Table 13.5 Values for PROBE= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses the probing strategy determined by PROC OPTMILP
0	NONE	Disables probing
1	MODERATE	Uses the probing moderately
2	AGGRESSIVE	Uses the probing aggressively

The default value is AUTOMATIC. See the section "Presolve and Probing" on page 646 for more information.

RELOBJGAP=*number*
specifies a stopping criterion based on the best integer objective (BestInteger) and the best bound on the objective function value (BestBound). The relative objective gap is equal to

$$| \text{BestInteger} - \text{BestBound} | / (1\text{E}{-10} + | \text{BestBound} |)$$

When this value becomes smaller than the specified gap size *number*, the procedure stops. The value of *number* can be any nonnegative number; the default value is 1E–4.

SCALE=*number* | *string*
indicates whether to scale the problem matrix. SCALE= can take either of the values AUTOMATIC (–1) and NONE (0). SCALE=AUTOMATIC scales the matrix as determined by PROC OPTMILP; SCALE=NONE disables scaling. The default value is AUTOMATIC.

SEED=*number*
specifies the initial seed of the random number generator. This option affects the perturbation in the simplex solvers; thus it might result in a different optimal solution and a different solver path. This option usually has a significant, but unpredictable, effect on the solution time. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of the seed is 100.

TARGET=*number*
specifies a stopping criterion for minimization (maximization) problems. If the best integer objective is better than or equal to *number*, the procedure stops. The value of *number* can be any number; the default value is the negative (positive) number that has the largest absolute value representable in your operating environment.

TIMETYPE=*number* | *string*
specifies whether CPU time or real time is used for the MAXTIME= option and the _OROPTMILP_ macro variable in a PROC OPTMILP call. Table 13.6 describes the valid values of the TIMETYPE= option.

Table 13.6 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

The default value of the TIMETYPE= option depends on the algorithm used and on various options. When the solver is used with distributed or multithreaded processing, then by default TIMETYPE= REAL. Otherwise, by default TIMETYPE= CPU. Table 13.7 describes the detailed logic for determining the default; the first context in the table that applies determines the default value. The NTHREADS= and NODES= options are specified in the PERFORMANCE statement. For more information about the NTHREADS= and NODES= options, see the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Table 13.7 Default Value for TIMETYPE= Option

Context	Default
NODES= value is nonzero for the decomposition algorithm	REAL
NTHREADS= value is greater than 1	REAL
NTHREADS= 1	CPU

Heuristics Option

HEURISTICS=*number* | *string*

controls the level of primal heuristics applied by PROC OPTMILP. This level determines how frequently primal heuristics are applied during the branch-and-bound tree search. It also affects the maximum number of iterations allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. The values of *string* and the corresponding values of *number* are listed in Table 13.8.

Table 13.8 Values for HEURISTICS= Option

number	*string*	Description
–1	AUTOMATIC	Applies the default level of heuristics, similar to MODERATE
0	NONE	Disables all primal heuristics
1	BASIC	Applies basic primal heuristics at low frequency
2	MODERATE	Applies most primal heuristics at moderate frequency
3	AGGRESSIVE	Applies all primal heuristics at high frequency

Setting HEURISTICS=NONE does not disable the heuristics that repair an infeasible input solution that is specified in a PRIMALIN= data set.

The default value of the HEURISTICS= option is AUTOMATIC. For details about primal heuristics, see the section "Primal Heuristics" on page 648.

Search Options

CONFLICTSEARCH=*number* | *string*

specifies the level of conflict search performed by PROC OPTMILP. Conflict search is used to find clauses resulting from infeasible subproblems that arise in the search tree. The values of *string* and the corresponding values of *number* are listed in Table 13.9.

Table 13.9 Values for CONFLICTSEARCH= Option

number	*string*	**Description**
–1	AUTOMATIC	Performs conflict search based on a strategy determined by PROC OPTMILP
0	NONE	Disables conflict search
1	MODERATE	Performs a moderate conflict search
2	AGGRESSIVE	Performs an aggressive conflict search

The default value is AUTOMATIC.

NODESEL=*number* | *string*
specifies the node selection strategy *string* or its corresponding value *number*, as listed in Table 13.10.

Table 13.10 Values for NODESEL= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses automatic node selection
0	BESTBOUND	Chooses the node with the best relaxed objective (best-bound-first strategy)
1	BESTESTIMATE	Chooses the node with the best estimate of the integer objective value (best-estimate-first strategy)
2	DEPTH	Chooses the most recently created node (depth-first strategy)

The default value is AUTOMATIC. For details about node selection, see the section "Node Selection" on page 645.

PRIORITY=0 | 1
indicates whether to use specified branching priorities for integer variables. PRIORITY=0 ignores variable priorities; PRIORITY=1 uses priorities when they exist. The default value is 1. See the section "Branching Priorities" on page 646 for details.

RESTARTS=*number* | *string*
specifies the strategy for restarting the processing of the root node. The values of *string* and the corresponding values of *number* are listed in Table 13.11.

Table 13.11 Values for RESTARTS= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses a restarting strategy determined by PROC OPTMILP
0	NONE	Disables restarting
1	BASIC	Uses a basic restarting strategy
2	MODERATE	Uses a moderate restarting strategy
3	AGGRESSIVE	Uses an aggressive restarting strategy

The default value is AUTOMATIC.

STRONGITER=*number* | **AUTOMATIC**

specifies the number of simplex iterations performed for each variable in the candidate list when using the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, PROC OPTMILP uses the default value; this value is calculated automatically.

STRONGLEN=*number* | **AUTOMATIC**

specifies the number of candidates used when performing the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, PROC OPTMILP uses the default value; this value is calculated automatically.

SYMMETRY=*number* | *string*

specifies the level of symmetry detection. Symmetry detection identifies groups of equivalent decision variables and uses this information to solve the problem more efficiently. The values of *string* and the corresponding values of *number* are listed in Table 13.12.

Table 13.12 Values for SYMMETRY= Option

number	*string*	**Description**
–1	AUTOMATIC	Performs symmetry detection based on a strategy that is determined by PROC OPTMILP
0	NONE	Disables symmetry detection
1	BASIC	Performs a basic symmetry detection
2	MODERATE	Performs a moderate symmetry detection
3	AGGRESSIVE	Performs an aggressive symmetry detection

The default value is AUTOMATIC. For more information about symmetry detection, see (Ostrowski 2008).

VARSEL=*number* | *string*

specifies the rule for selecting the branching variable. The values of *string* and the corresponding values of *number* are listed in Table 13.13.

Table 13.13 Values for VARSEL= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses automatic branching variable selection
0	MAXINFEAS	Chooses the variable with maximum infeasibility
1	MININFEAS	Chooses the variable with minimum infeasibility
2	PSEUDO	Chooses a branching variable based on pseudocost
3	STRONG	Uses strong branching variable selection strategy

The default value is AUTOMATIC. For details about variable selection, see the section "Variable Selection" on page 645.

Cut Options

Table 13.14 describes the *string* and *number* values for the cut options in PROC OPTMILP.

Table 13.14 Values for Individual Cut Options

number	*string*	**Description**
–1	AUTOMATIC	Generates cutting planes based on a strategy determined by PROC OPTMILP
0	NONE	Disables generation of cutting planes
1	MODERATE	Uses a moderate cut strategy
2	AGGRESSIVE	Uses an aggressive cut strategy

You can specify the CUTSTRATEGY= option to set the overall aggressiveness of the cut generation in PROC OPTMILP. Alternatively, you can use the ALLCUTS= option to set all cut types to the same level. You can override the ALLCUTS= value by using the options that correspond to particular cut types. For example, if you want PROC OPTMILP to generate only Gomory cuts, specify ALLCUTS=NONE and CUTGOMORY=AUTOMATIC. If you want to generate all cuts aggressively but generate no lift-and-project cuts, set ALLCUTS=AGGRESSIVE and CUTLAP=NONE.

ALLCUTS=*number* | *string*
provides a shorthand way of setting all the cuts-related options in one setting. In other words, ALLCUTS=*number* is equivalent to setting each of the individual cuts parameters to the same value *number*. Thus, ALLCUTS=–1 has the effect of setting CUTCLIQUE=–1, CUTFLOWCOVER=–1, CUTFLOWPATH=–1, . . . , CUTMULTICOMMODITY=–1, and CUTZEROHALF=–1. Table 13.14 lists the values that can be assigned to *string* and *number*. In addition, you can override levels for individual cuts with the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIMPLIED=, CUTKNAPSACK=, CUTLAP=, CUTMILIFTED=, CUTMIR=, CUTMULTICOMMODITY=, and CUTZEROHALF= options. If the ALLCUTS= option is not specified, all the cuts-related options are either set to their individually specified values (if the corresponding option is specified) or to their default values (if that option is not specified).

CUTCLIQUE=*number* | *string*
specifies the level of clique cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTCLIQUE= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTFLOWCOVER=*number* | *string*
specifies the level of flow cover cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTFLOWCOVER= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTFLOWPATH=*number* | *string*
specifies the level of flow path cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTFLOWPATH= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTGOMORY=*number* | *string*
specifies the level of Gomory cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTGOMORY= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTGUB=*number* | *string*
specifies the level of generalized upper bound (GUB) cover cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTGUB= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTIMPLIED=*number* | *string*
specifies the level of implied bound cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTIMPLIED= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTKNAPSACK=*number* | *string*
specifies the level of knapsack cover cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTKNAPSACK= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTLAP=*number* | *string*
specifies the level of lift-and-project (LAP) cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTLAP= option overrides the ALLCUTS= option. The default value is NONE.

CUTMILIFTED=*number* | *string*
specifies the level of mixed lifted 0-1 cuts that are generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *option* and *num*. The CUTMILIFTED= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTMIR=*number* | *string*
specifies the level of mixed integer rounding (MIR) cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTMIR= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTMULTICOMMODITY=*number* | *string*
specifies the level of multicommodity network flow cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTMULTICOMMODITY= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTSFACTOR=*number*
specifies a row multiplier factor for cuts. The number of cuts that are added is limited to *number* times the original number of rows. The value of *number* can be any nonnegative number less than or equal to 100; the default value is automatically calculated by PROC OPTMILP.

CUTSTRATEGY=*number* | *string*

CUTS=*number* | *string*
specifies the overall aggressiveness of the cut generation in the solver. Setting a nondefault value adjusts a number of cut parameters such that the cut generation is basic, moderate, or aggressive compared to the default value.

CUTZEROHALF=*number* | *string*
specifies the level of zero-half cuts that are generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*.The CUTZEROHALF= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

Decomposition Algorithm Statements

The following statements are available for the decomposition algorithm in the OPTMILP procedure:

DECOMP < *options* > **;**

DECOMP_MASTER < *options* > **;**

DECOMP_MASTER_IP < *options* > **;**

DECOMP_SUBPROB < *options* > **;**

For more information about these statements, see Chapter 15, "The Decomposition Algorithm."

PERFORMANCE Statement

PERFORMANCE < *performance-options* > **;**

The PERFORMANCE statement specifies *performance-options* for single-machine mode and distributed mode, and requests detailed performance results of the OPTMILP procedure.

You can also use the PERFORMANCE statement to control whether the OPTMILP procedure executes in single-machine or distributed mode. The decomposition algorithm and the option tuner can be run in both single-machine and distributed modes. The parallel branch-and-cut algorithm can be run only in single-machine mode.

The PERFORMANCE statement is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

NOTE: Distributed mode requires SAS High-Performance Optimization.

TUNER Statement

TUNER < *performance-options* > **;**

The TUNER statement invokes the OPTMILP option tuner. The option tuner is a tool that enables you to explore alternative (and potentially better) option configurations for your optimization problems. For more information about this feature, see Chapter 16, "The OPTMILP Option Tuner."

Details: OPTMILP Procedure

Data Input and Output

This subsection describes the PRIMALIN= data set required to warm start PROC OPTMILP, in addition to the PRIMALOUT= and DUALOUT= data sets.

Definitions of Variables in the PRIMALIN= Data Set

The PRIMALIN= data set has two required variables defined as follows:

VAR
specifies the variable (column) names of the problem. The values should match the column names in the DATA= data set for the current problem.

VALUE
specifies the solution value for each variable in the problem.

NOTE: If PROC OPTMILP produces a feasible solution, the primal output data set from that run can be used as the PRIMALIN= data set for a subsequent run, provided that the variable names are the same. If this input solution is not feasible for the subsequent run, the solver automatically tries to repair it. See the section "Warm Start" on page 643 for more details.

Definitions of Variables in the PRIMALOUT= Data Set

PROC OPTMILP stores the current best integer feasible solution of the problem in the data set specified by the PRIMALOUT= option. The variables in this data set are defined as follows:

_OBJ_ID_
specifies the identifier of the objective function.

_RHS_ID_
specifies the identifier of the right-hand side.

VAR
specifies the variable (column) names.

TYPE
specifies the variable type. _TYPE_ can take one of the following values:

C	continuous variable
I	general integer variable
B	binary variable (0 or 1)

OBJCOEF
specifies the coefficient of the variable in the objective function.

LBOUND
specifies the lower bound on the variable.

UBOUND
specifies the upper bound on the variable.

VALUE
specifies the value of the variable in the current solution.

Definitions of the DUALOUT= Data Set Variables

The DUALOUT= data set contains the constraint activities that correspond to the primal solution in the PRIMALOUT= data set. Information about additional objective rows of the MILP problem is not included. The variables in this data set are defined as follows:

_OBJ_ID_
specifies the identifier of the objective function from the input data set.

_RHS_ID_
specifies the identifier of the right-hand side from the input data set.

ROW
specifies the constraint (row) name.

TYPE
specifies the constraint type. _TYPE_ can take one of the following values:

L "less than or equal" constraint

E equality constraint

G "greater than or equal" constraint

R ranged constraint (both "less than or equal" and "greater than or equal")

RHS
specifies the value of the right-hand side of the constraint. It takes a missing value for a ranged constraint.

_L_RHS_
specifies the lower bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_U_RHS_
specifies the upper bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

ACTIVITY
specifies the activity of a constraint for a given primal solution. In other words, the value of _ACTIVITY_ for the ith constraint is equal to $\mathbf{a}_i^T \mathbf{x}$, where $\mathbf{a}_i$ refers to the ith row of the constraint matrix and $\mathbf{x}$ denotes the vector of the current primal solution.

Warm Start

PROC OPTMILP enables you to input a warm start solution by using the PRIMALIN= option. PROC OPTMILP checks that the decision variables named in _VAR_ are the same as those in the MPS-format SAS data set. If they are not the same, PROC OPTMILP issues a warning and ignores the input solution. PROC OPTMILP also checks whether the solution is infeasible, contains missing values, or contains fractional values for integer variables. If this is the case, PROC OPTMILP attempts to repair the solution with a number of specialized repair heuristics. The success of the attempt largely depends both on the specific model and on the proximity between the input solution and an integer feasible solution. An infeasible input solution can be considered a hint for PROC OPTMILP that might or might not help to solve the problem.

An integer feasible or repaired input solution provides an incumbent solution in addition to an upper (min) or lower (max) bound for the branch-and-bound algorithm. PROC OPTMILP uses the input solution to reduce the search space and to guide the search process. When it is difficult to find a good integer feasible solution for a problem, warm start can reduce solution time significantly.

Branch-and-Bound Algorithm

The branch-and-bound algorithm, first proposed by Land and Doig (1960), is an effective approach to solving mixed integer linear programs. The following discussion outlines the approach and explains how PROC OPTMILP enhances the basic algorithm by using several advanced techniques.

The branch-and-bound algorithm solves a mixed integer linear program by dividing the search space and generating a sequence of subproblems. The search space of a mixed integer linear program can be represented by a tree. Each node in the tree is identified with a subproblem derived from previous subproblems on the path that leads to the root of the tree. The subproblem (MILP^0) associated with the root is identical to the original problem, which is called (MILP), given in the section "Overview: OPTMILP Procedure" on page 625.

The linear programming relaxation (LP^0) of (MILP^0) can be written as

$$\begin{array}{rl} \min & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{array}$$

The branch-and-bound algorithm generates subproblems along the nodes of the tree by using the following scheme. Consider $\bar{x}^0$, the optimal solution to (LP^0), which is usually obtained by using the dual simplex algorithm. If $\bar{x}_i^0$ is an integer for all $i \in \mathcal{S}$, then $\bar{x}^0$ is an optimal solution to (MILP). Suppose that for some $i \in \mathcal{S}$, $\bar{x}_i^0$ is nonintegral. In that case the algorithm defines two new subproblems (MILP^1) and (MILP^2), descendants of the parent subproblem (MILP^0). The subproblem (MILP^1) is identical to (MILP^0) except for the additional constraint

$$x_i \leq \lfloor \bar{x}_i^0 \rfloor$$

and the subproblem (MILP^2) is identical to (MILP^0) except for the additional constraint

$$x_i \geq \lceil \bar{x}_i^0 \rceil$$

The notation $\lfloor y \rfloor$ represents the largest integer that is less than or equal to y, and the notation $\lceil y \rceil$ represents the smallest integer that is greater than or equal to y. The two preceding constraints can be handled by modifying the bounds of the variable x_i rather than by explicitly adding the constraints to the constraint matrix. The two new subproblems do not have $\bar{x}^0$ as a feasible solution, but the integer solution to (MILP) must satisfy one of the preceding constraints. The two subproblems thus defined are called *active nodes* in the branch-and-bound tree, and the variable x_i is called the *branching variable*.

In the next step the branch-and-bound algorithm chooses one of the active nodes and attempts to solve the linear programming relaxation of that subproblem. The relaxation might be infeasible, in which case the subproblem is dropped (fathomed). If the subproblem can be solved and the solution is *integer feasible* (that is, x_i is an integer for all $i \in \mathcal{S}$), then its objective value provides an *upper bound* for the objective value in the minimization problem (MILP); if the solution is not integer feasible, then it defines two new subproblems. Branching continues in this manner until there are no active nodes. At this point the best integer solution found is an optimal solution for (MILP). If no integer solution has been found, then (MILP) is integer infeasible. You can specify other criteria to stop the branch-and-bound algorithm before it processes all the active nodes; see the section "Controlling the Branch-and-Bound Algorithm" on page 644 for details.

Upper bounds from integer feasible solutions can be used to *fathom* or *cut off* active nodes. Since the objective value of an optimal solution cannot be greater than an upper bound, active nodes with lower bounds higher than an existing upper bound can be safely deleted. In particular, if z is the objective value of the current best integer solution, then any active subproblems whose relaxed objective value is greater than or equal to z can be discarded.

It is important to realize that mixed integer linear programs are non-deterministic polynomial-time hard (NP-hard). Roughly speaking, this means that the effort required to solve a mixed integer linear program grows exponentially with the size of the problem. For example, a problem with 10 binary variables can generate in the worst case $2^{10} = 1,024$ nodes in the branch-and-bound tree. A problem with 20 binary variables can generate in the worst case $2^{20} = 1,048,576$ nodes in the branch-and-bound tree. Although it is unlikely that the branch-and-bound algorithm has to generate every single possible node, the need to explore even a small fraction of the potential number of nodes for a large problem can be resource-intensive.

A number of techniques can speed up the search progress of the branch-and-bound algorithm. Heuristics are used to find feasible solutions, which can improve the upper bounds on solutions of mixed integer linear programs. Cutting planes can reduce the search space and thus improve the lower bounds on solutions of mixed integer linear programs. When using cutting planes, the branch-and-bound algorithm is also called the *branch-and-cut algorithm*. Preprocessing can reduce problem size and improve problem solvability. PROC OPTMILP employs various heuristics, cutting planes, preprocessing, and other techniques, which you can control through corresponding options.

Controlling the Branch-and-Bound Algorithm

There are numerous strategies that can be used to control the branch-and-bound search (see Linderoth and Savelsbergh 1998, Achterberg, Koch, and Martin 2005). PROC OPTMILP implements the most widely used strategies and provides several options that enable you to direct the choice of the next active node and of the branching variable. In the discussion that follows, let (LP^k) be the linear programming relaxation of subproblem (MILP^k). Also, let

$$f_i(k) = \bar{x}_i^k - \lfloor \bar{x}_i^k \rfloor$$

where $\bar{x}^k$ is the optimal solution to the relaxation problem (LP^k) solved at node k.

Node Selection

The NODESEL= option specifies the strategy used to select the next active node. The valid keywords for this option are AUTOMATIC, BESTBOUND, BESTESTIMATE, and DEPTH. The following list describes the strategy associated with each keyword:

AUTOMATIC — allows PROC OPTMILP to choose the best node selection strategy based on problem characteristics and search progress. This is the default setting.

BESTBOUND — chooses the node with the smallest (or largest, in the case of a maximization problem) relaxed objective value. The best-bound strategy tends to reduce the number of nodes to be processed and can improve lower bounds quickly. However, if there is no good upper bound, the number of active nodes can be large. This can result in the solver running out of memory.

BESTESTIMATE — chooses the node with the smallest (or largest, in the case of a maximization problem) objective value of the estimated integer solution. Besides improving lower bounds, the best-estimate strategy also attempts to process nodes that can yield good feasible solutions.

DEPTH — chooses the node that is deepest in the search tree. Depth-first search is effective in locating feasible solutions, since such solutions are usually deep in the search tree. Compared to the costs of the best-bound and best-estimate strategies, the cost of solving LP relaxations is less in the depth-first strategy. The number of active nodes is generally small, but it is possible that the depth-first search will remain in a portion of the search tree with no good integer solutions. This occurrence is computationally expensive.

Variable Selection

The VARSEL= option specifies the strategy used to select the next branching variable. The valid keywords for this option are AUTOMATIC, MAXINFEAS, MININFEAS, PSEUDO, and STRONG. The following list describes the action taken in each case when $\bar{x}^k$, a relaxed optimal solution of (MILP^k), is used to define two active subproblems. In the following list, "INTTOL" refers to the value assigned using the INTTOL= option. For details about the INTTOL= option, see the section "Control Options" on page 631.

AUTOMATIC — enables PROC OPTMILP to choose the best variable selection strategy based on problem characteristics and search progress. This is the default setting.

MAXINFEAS — chooses as the branching variable the variable x_i such that i maximizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and}$$

$$\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

MININFEAS — chooses as the branching variable the variable x_i such that i minimizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and}$$

$$\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

PSEUDO chooses as the branching variable the variable x_i such that i maximizes the weighted up and down pseudocosts. Pseudocost branching attempts to branch on significant variables first, quickly improving lower bounds. Pseudocost branching estimates significance based on historical information; however, this approach might not be accurate for future search.

STRONG chooses as the branching variable the variable x_i such that i maximizes the estimated improvement in the objective value. Strong branching first generates a list of candidates, then branches on each candidate and records the improvement in the objective value. The candidate with the largest improvement is chosen as the branching variable. Strong branching can be effective for combinatorial problems, but it is usually computationally expensive.

Branching Priorities

In some cases, it is possible to speed up the branch-and-bound algorithm by branching on variables in a specific order. You can accomplish this in PROC OPTMILP by attaching branching priorities to the integer variables in your model.

You can set branching priorities for use by PROC OPTMILP in two ways. You can specify the branching priorities directly in the input MPS-format data set; see the section "BRANCH Section (Optional)" on page 847 for details. If you are constructing a model in PROC OPTMODEL, you can set branching priorities for integer variables by using the .priority suffix. More information about this suffix is available in the section "Integer Variable Suffixes" on page 137 in Chapter 5. For an example in which branching priorities are used, see Example 8.3.

Presolve and Probing

PROC OPTMILP includes a variety of presolve techniques to reduce problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix. Presolve might also modify the constraint coefficients to tighten the formulation of the problem.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels by specifying the PRESOLVER= option.

Probing is a technique that tentatively sets each binary variable to 0 or 1, then explores the logical consequences (Savelsbergh 1994). Probing can expedite the solution of a difficult problem by fixing variables and improving the model. However, probing is often computationally expensive and can significantly increase the solution time in some cases. You can enable probing at different levels by specifying the PROBE= option.

Cutting Planes

The feasible region of every linear program forms a *polyhedron*. Every polyhedron in n-space can be written as a finite number of half-spaces (equivalently, inequalities). In the notation used in this chapter, this polyhedron is defined by the set $\mathcal{Q} = \{x \in \mathbb{R}^n \mid Ax \leq b, l \leq x \leq u\}$. After you add the restriction that

some variables must be integral, the set of feasible solutions, $\mathcal{F} = \{x \in \mathcal{Q} \mid x_i \in \mathbb{Z} \ \forall i \in \mathcal{S}\}$, no longer forms a polyhedron.

The *convex hull* of a set X is the minimal convex set that contains X. In solving a mixed integer linear program, in order to take advantage of LP-based algorithms you want to find the convex hull, $\text{conv}(\mathcal{F})$, of $\mathcal{F}$. If you can find $\text{conv}(\mathcal{F})$ and describe it compactly, then you can solve a mixed integer linear program with a linear programming solver. This is generally very difficult, so you must be satisfied with finding an approximation. Typically, the better the approximation, the more efficiently the LP-based branch-and-bound algorithm can perform.

As described in the section "Branch-and-Bound Algorithm" on page 643, the branch-and-bound algorithm begins by solving the linear programming relaxation over the polyhedron $\mathcal{Q}$. Clearly, $\mathcal{Q}$ contains the convex hull of the feasible region of the original integer program; that is, $\text{conv}(\mathcal{F}) \subseteq \mathcal{Q}$.

Cutting plane techniques are used to tighten the linear relaxation to better approximate $\text{conv}(\mathcal{F})$. Assume you are given a solution $\bar{x}$ to some intermediate linear relaxation during the branch-and-bound algorithm. A cut, or valid inequality ($\pi x \leq \pi^0$), is some half-space with the following characteristics:

- The half-space contains $\text{conv}(\mathcal{F})$; that is, every integer feasible solution is feasible for the cut ($\pi x \leq \pi^0, \forall x \in \mathcal{F}$).
- The half-space does not contain the current solution $\bar{x}$; that is, $\bar{x}$ is not feasible for the cut ($\pi \bar{x} > \pi^0$).

Cutting planes were first made popular by Dantzig, Fulkerson, and Johnson (1954) in their work on the traveling salesman problem. The two major classifications of cutting planes are *generic cuts* and *structured cuts*. Generic cuts are based solely on algebraic arguments and can be applied to any relaxation of any integer program. Structured cuts are specific to certain structures that can be found in some relaxations of the mixed integer linear program. These structures are automatically discovered during the cut initialization phase of PROC OPTMILP. Table 13.15 lists the various types of cutting planes that are built into PROC OPTMILP. Included in each type are algorithms for numerous variations based on different relaxations and lifting techniques. For a survey of cutting plane techniques for mixed integer programming, see Marchand et al. (1999). For a survey of lifting techniques, see Atamturk (2004).

Table 13.15 Cutting Planes in PROC OPTMILP

Generic Cutting Planes	Structured Cutting Planes
Gomory mixed integer	Cliques
Lift-and-project	Flow cover
Mixed integer rounding	Flow path
Mixed lifted 0-1	Generalized upper bound cover
Zero-half	Implied bound
	Knapsack cover
	Multicommodity network flow

You can set levels for individual cuts by using the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIMPLIED=, CUTKNAPSACK=, CUTLAP=, CUTMIR=, CUTMULTICOMMODITY=, and CUTZEROHALF= options. The valid levels for these options are given in Table 13.14.

The cut level determines the internal strategy used by PROC OPTMILP for generating the cutting planes. The strategy consists of several factors, including how frequently the cut search is called, the number of cuts allowed, and the aggressiveness of the search algorithms.

Sophisticated cutting planes, such as those included in PROC OPTMILP, can take a great deal of CPU time. Usually, the additional tightening of the relaxation helps speed up the overall process because it provides better bounds for the branch-and-bound tree and helps guide the LP solver toward integer solutions. In rare cases, shutting off cutting planes completely might lead to faster overall run times.

The default settings of PROC OPTMILP have been tuned to work well for most instances. However, problem-specific expertise might suggest adjusting one or more of the strategies. These options give you that flexibility.

Primal Heuristics

Primal heuristics, an important component of PROC OPTMILP, are applied during the branch-and-bound algorithm. They are used to find integer feasible solutions early in the search tree, thereby improving the upper bound for a minimization problem. Primal heuristics play a role that is complementary to cutting planes in reducing the gap between the upper and lower bounds, thus reducing the size of the branch-and-bound tree.

Applying primal heuristics in the branch-and-bound algorithm assists in the following areas:

- finding a good upper bound early in the tree search (this can lead to earlier fathoming, resulting in fewer subproblems to be processed)
- locating a reasonably good feasible solution when that is sufficient (sometimes a good feasible solution is the best the solver can produce within certain time or resource limits)
- providing upper bounds for some bound-tightening techniques

The OPTMILP procedure implements several heuristic methodologies. Some algorithms, such as rounding and iterative rounding (diving) heuristics, attempt to construct an integer feasible solution by using fractional solutions to the continuous relaxation at each node of the branch-and-cut tree. Other algorithms start with an incumbent solution and attempt to find a better solution within a neighborhood of the current best solution.

The HEURISTICS= option enables you to control the level of primal heuristics that are applied by PROC OPTMILP. This level determines how frequently primal heuristics are applied during the tree search. Some expensive heuristics might be disabled by the solver at less aggressive levels. Setting the HEURISTICS= option to a lower level also reduces the maximum number of iterations that are allowed in iterative heuristics. The valid values for this option are listed in Table 13.8.

Parallel Processing

You can run the branch-and-cut algorithm only in single-machine mode. In single-machine mode, the computation is executed by multiple threads on a single computer.

You can run the decomposition algorithm and option tuner in either single-machine or distributed mode. In distributed mode, the computation is executed on multiple computing nodes in a distributed computing environment.

NOTE: Distributed mode requires SAS High-Performance Optimization.

You can specify options that control parallel processing in the PERFORMANCE statement, which is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Node Log

The following information about the status of the branch-and-bound algorithm is printed in the node log:

Node	indicates the sequence number of the current node in the search tree.
Active	indicates the current number of active nodes in the branch-and-bound tree.
Sols	indicates the number of feasible solutions found so far.
BestInteger	indicates the best upper bound (assuming minimization) found so far.
BestBound	indicates the best lower bound (assuming minimization) found so far.
Gap	indicates the relative gap between BestInteger and BestBound, displayed as a percentage. If the relative gap is larger than 1,000, then the absolute gap is displayed. If no active nodes remain, the value of Gap is 0.
Time	indicates the elapsed real time.

The LOGFREQ= and LOGLEVEL= options can be used to control the amount of information printed in the node log. By default a new entry is included in the log at the first node, at the last node, and at 100-node intervals. A new entry is also included each time a better integer solution is found. The LOGFREQ= option enables you to change the interval between entries in the node log. Figure 13.4 shows a sample node log.

Figure 13.4 Sample Node Log

```
NOTE: The problem ex1data has 10 variables (0 binary, 10 integer, 0 free, 0
      fixed).
NOTE: The problem has 2 constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 20 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 2 variables and 0 constraints.
NOTE: The MILP presolver removed 4 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 8 variables, 2 constraints, and 16 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       3     85.0000000    158.0000000   46.20%       0
             0       1       3     85.0000000     88.0955497    3.51%       0
             0       1       3     85.0000000     88.0955497    3.51%       0
             0       1       3     85.0000000     88.0955497    3.51%       0
NOTE: The MILP presolver is applied again.
             0       1       3     85.0000000     88.0955497    3.51%       0
             0       1       3     85.0000000     88.0626822    3.48%       0
             0       1       3     85.0000000     87.8820655    3.28%       0
             0       1       3     85.0000000     87.8539763    3.25%       0
             0       1       3     85.0000000     87.7208690    3.10%       0
             0       1       3     85.0000000     87.7180302    3.10%       0
             0       1       3     85.0000000     87.7133502    3.09%       0
             0       1       3     85.0000000     87.7128245    3.09%       0
NOTE: The MILP solver added 2 cuts with 8 cut coefficients at the root.
             3       1       4     86.0000000     87.3854749    1.59%       0
             4       0       5     87.0000000     87.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 87.
NOTE: The data set WORK.EX1SOLN has 10 observations and 8 variables.
```

ODS Tables

PROC OPTMILP creates three Output Delivery System (ODS) tables by default. The first table, ProblemSummary, is a summary of the input MILP problem. The second table, SolutionSummary, is a brief summary of the solution status. The third table, PerformanceInfo, is a summary of performance options. You can use ODS table names to select tables and create output data sets. For more information about ODS, see *SAS Output Delivery System: User's Guide*.

If you specify a value of 2 for the PRINTLEVEL= option, then the ProblemStatistics table is produced. This table contains information about the problem data. See the section "Problem Statistics" on page 654 for more information.

If you specify the DETAILS option in the PERFORMANCE statement, then the Timing table is produced.

Table 13.16 lists all the ODS tables that can be produced by the OPTMILP procedure, along with the statement and option specifications required to produce each table.

Table 13.16 ODS Tables Produced by PROC OPTMILP

ODS Table Name	Description	Statement	Option
ProblemSummary	Summary of the input MILP problem	PROC OPTMILP	PRINTLEVEL=1 (default)
SolutionSummary	Summary of the solution status	PROC OPTMILP	PRINTLEVEL=1 (default)
ProblemStatistics	Description of input problem data	PROC OPTMILP	PRINTLEVEL=2
PerformanceInfo	List of performance options and their values	PROC OPTMILP	PRINTLEVEL=1 (default)
Timing	Detailed solution timing	PERFORMANCE	DETAILS

A typical ProblemSummary table is shown in Figure 13.5.

Figure 13.5 Example PROC OPTMILP Output: Problem Summary

The OPTMILP Procedure

Problem Summary	
Problem Name	EX_MIP
Objective Sense	Minimization
Objective Function	COST
RHS	RHS
Number of Variables	3
Bounded Above	0
Bounded Below	0
Bounded Above and Below	3
Free	0
Fixed	0
Binary	3
Integer	0
Number of Constraints	3
LE (<=)	2
EQ (=)	0
GE (>=)	1
Range	0
Constraint Coefficients	8

A typical SolutionSummary table is shown in Figure 13.6.

Figure 13.6 Example PROC OPTMILP Output: Solution Summary

The OPTMILP Procedure

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	COST
Solution Status	Optimal
Objective Value	-7
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	0
Best Bound	-7
Nodes	0
Iterations	0
Presolve Time	0.00
Solution Time	0.00

You can create output data sets from these tables by using the ODS OUTPUT statement. The output data sets from the preceding example are displayed in Figure 13.7 and Figure 13.8, where you can also find variable names for the tables used in the ODS template of the OPTMILP procedure.

Figure 13.7 ODS Output Data Set: Problem Summary

Problem Summary

Obs	Label1	cValue1	nValue1
1	Problem Name	EX_MIP	.
2	Objective Sense	Minimization	.
3	Objective Function	COST	.
4	RHS	RHS	.
5			.
6	Number of Variables	3	3.000000
7	Bounded Above	0	0
8	Bounded Below	0	0
9	Bounded Above and Below	3	3.000000
10	Free	0	0
11	Fixed	0	0
12	Binary	3	3.000000
13	Integer	0	0
14			.
15	Number of Constraints	3	3.000000
16	LE (<=)	2	2.000000
17	EQ (=)	0	0
18	GE (>=)	1	1.000000
19	Range	0	0
20			.
21	Constraint Coefficients	8	8.000000

Figure 13.8 ODS Output Data Set: Solution Summary

Solution Summary

Obs	Label1	cValue1	nValue1
1	Solver	MILP	.
2	Algorithm	Branch and Cut	.
3	Objective Function	COST	.
4	Solution Status	Optimal	.
5	Objective Value	-7	-7.000000
6			.
7	Relative Gap	0	0
8	Absolute Gap	0	0
9	Primal Infeasibility	0	0
10	Bound Infeasibility	0	0
11	Integer Infeasibility	0	0
12			.
13	Best Bound	-7	-7.000000
14	Nodes	0	0
15	Iterations	0	0
16	Presolve Time	0.00	0.001000
17	Solution Time	0.00	0.001000

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMILP procedure causes the ODS table ProblemStatistics to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 13.9 demonstrates the contents of the ODS table ProblemStatistics.

Figure 13.9 ODS Table ProblemStatistics

ProblemStatistics

Obs	Label1	cValue1	nValue1
1	Number of Constraint Matrix Nonzeros	8	8.000000
2	Maximum Constraint Matrix Coefficient	3	3.000000
3	Minimum Constraint Matrix Coefficient	1	1.000000
4	Average Constraint Matrix Coefficient	1.875	1.875000
5			.
6	Number of Objective Nonzeros	3	3.000000
7	Maximum Objective Coefficient	4	4.000000
8	Minimum Objective Coefficient	2	2.000000
9	Average Objective Coefficient	3	3.000000
10			.
11	Number of RHS Nonzeros	3	3.000000
12	Maximum RHS	7	7.000000
13	Minimum RHS	4	4.000000
14	Average RHS	5.3333333333	5.333333
15			.
16	Maximum Number of Nonzeros per Column	3	3.000000
17	Minimum Number of Nonzeros per Column	2	2.000000
18	Average Number of Nonzeros per Column	2.67	2.666667
19			.
20	Maximum Number of Nonzeros per Row	3	3.000000
21	Minimum Number of Nonzeros per Row	2	2.000000
22	Average Number of Nonzeros per Row	2.67	2.666667

The variable names in the ODS table ProblemStatistics are Label1, cValue1, and nValue1.

Macro Variable _OROPTMILP_

The OPTMILP procedure defines a macro variable named _OROPTMILP_. This variable contains a character string that indicates the status of the OPTMILP procedure upon termination. The various terms of the variable are interpreted as follows.

STATUS

indicates the solver status at termination. It can take one of the following values:

OK	The procedure terminated normally.
SYNTAX_ERROR	Incorrect syntax was used.
DATA_ERROR	The input data was inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem occurred in reading or writing data.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take one of the following values:

BAC	The branch-and-cut algorithm produced the solution data.
DECOMP	The decomposition algorithm produced the solution data.

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
OPTIMAL_AGAP	The solution is optimal within the absolute gap specified by the ABSOBJGAP= option.
OPTIMAL_RGAP	The solution is optimal within the relative gap specified by the RELOBJGAP= option.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances due to scaling or choice of a small INTTOL= value.
TARGET	The solution is not worse than the target specified by the TARGET= option.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
SOLUTION_LIM	The solver reached the maximum number of solutions specified by the MAXSOLS= option.
NODE_LIM_SOL	The solver reached the maximum number of nodes specified by the MAXNODES= option and found a solution.
NODE_LIM_NOSOL	The solver reached the maximum number of nodes specified by the MAXNODES= option and did not find a solution.
TIME_LIM_SOL	The solver reached the execution time limit specified by the MAXTIME= option and found a solution.

TIME_LIM_NOSOL	The solver reached the execution time limit specified by the MAXTIME= option and did not find a solution.
ABORT_SOL	The solver was stopped by the user but still found a solution.
ABORT_NOSOL	The solver was stopped by the user and did not find a solution.
OUTMEM_SOL	The solver ran out of memory but still found a solution.
OUTMEM_NOSOL	The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
FAIL_SOL	The solver stopped due to errors but still found a solution.
FAIL_NOSOL	The solver stopped due to errors and did not find a solution.

OBJECTIVE
indicates the objective value obtained by the solver at termination.

RELATIVE_GAP
indicates the relative gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the OPTMILP procedure. The relative gap is equal to

$$| \text{BestInteger} - \text{BestBound} | / (1\text{E}{-10} + | \text{BestBound} |)$$

ABSOLUTE_GAP
indicates the absolute gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the OPTMILP procedure. The absolute gap is equal to $| \text{BestInteger} - \text{BestBound} |$.

PRIMAL_INFEASIBILITY
indicates the maximum (absolute) violation of the primal constraints by the solution.

BOUND_INFEASIBILITY
indicates the maximum (absolute) violation by the solution of the lower or upper bounds (or both).

INTEGER_INFEASIBILITY
indicates the maximum (absolute) violation of the integrality of integer variables returned by the OPTMILP procedure.

BEST_BOUND
indicates the best bound on the objective function value at termination. A missing value indicates that the OPTMILP procedure was not able to obtain such a bound.

NODES
indicates the number of nodes enumerated by the OPTMILP procedure when using the branch-and-bound algorithm.

ITERATIONS
indicates the number of simplex iterations taken to solve the problem.

PRESOLVE_TIME
indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

Examples: OPTMILP Procedure

This section contains examples that illustrate the options and syntax of PROC OPTMILP. Example 13.1 demonstrates a model contained in an MPS-format SAS data set and finds an optimal solution by using PROC OPTMILP. Example 13.2 illustrates the use of standard MPS files in PROC OPTMILP. Example 13.3 demonstrates how to warm start PROC OPTMILP. More detailed examples of mixed integer linear programs, along with example SAS code, are given in Chapter 8.

Example 13.1: Simple Integer Linear Program

This example illustrates a model in an MPS-format SAS data set. This data set is passed to PROC OPTMILP, and a solution is found.

Consider a scenario where you have a container with a set of limiting attributes (volume V and weight W) and a set I of items that you want to pack. Each item type i has a certain value p_i, a volume v_i, and a weight w_i. You must choose at most four items of each type so that the total value is maximized and all the chosen items fit into the container. Let x_i be the number of items of type i to be included in the container. This model can be formulated as the following integer linear program:

$$
\begin{array}{lllll}
\max & \displaystyle\sum_{i \in I} p_i x_i & & & \\
\text{s.t.} & \displaystyle\sum_{i \in I} v_i x_i & \leq & V & \text{(volume\_con)} \\
& \displaystyle\sum_{i \in I} w_i x_i & \leq & W & \text{(weight\_con)} \\
& x_i & \leq & 4 \quad \forall i \in I & \\
& x_i \in \mathbb{Z}^+ & & \forall i \in I &
\end{array}
$$

Constraint (volume_con) enforces the volume capacity limit, while constraint (weight_con) enforces the weight capacity limit. An instance of this problem can be saved in an MPS-format SAS data set by using the following code:

```
data ex1data;
    input field1 $ field2 $ field3 $ field4 field5 $ field6;
    datalines;
NAME        .             ex1data          .     .                .
ROWS        .             .                .     .                .
MAX         z             .                .     .                .
L           volume_con    .                .     .                .
L           weight_con    .                .     .                .
COLUMNS     .             .                .     .                .
.           .MRK0         'MARKER'         .     'INTORG'         .
.           x[1]          z                1     volume_con       10
.           x[1]          weight_con       12    .                .
.           x[2]          z                2     volume_con       300
.           x[2]          weight_con       15    .                .
.           x[3]          z                3     volume_con       250
.           x[3]          weight_con       72    .                .
.           x[4]          z                4     volume_con       610
.           x[4]          weight_con       100   .                .
.           x[5]          z                5     volume_con       500
.           x[5]          weight_con       223   .                .
.           x[6]          z                6     volume_con       120
.           x[6]          weight_con       16    .                .
.           x[7]          z                7     volume_con       45
.           x[7]          weight_con       73    .                .
.           x[8]          z                8     volume_con       100
.           x[8]          weight_con       12    .                .
.           x[9]          z                9     volume_con       200
.           x[9]          weight_con       200   .                .
.           x[10]         z                10    volume_con       61
.           x[10]         weight_con       110   .                .
.           .MRK1         'MARKER'         .     'INTEND'         .
RHS         .             .                .     .                .
.           .RHS.         volume_con       1000  .                .
.           .RHS.         weight_con       500   .                .
BOUNDS      .             .                .     .                .
UP          .BOUNDS.      x[1]             4     .                .
UP          .BOUNDS.      x[2]             4     .                .
UP          .BOUNDS.      x[3]             4     .                .
UP          .BOUNDS.      x[4]             4     .                .
UP          .BOUNDS.      x[5]             4     .                .
UP          .BOUNDS.      x[6]             4     .                .
UP          .BOUNDS.      x[7]             4     .                .
UP          .BOUNDS.      x[8]             4     .                .
UP          .BOUNDS.      x[9]             4     .                .
UP          .BOUNDS.      x[10]            4     .                .
ENDATA      .             .                .     .                .
;
```

In the COLUMNS section of this data set, the name of the objective is z, and the objective coefficients p_i appear in field4. The coefficients v_i of (volume_con) appear in field6. The coefficients w_i of (weight_con) appear in field4. In the RHS section, the bounds V and W appear in field4.

This problem can be solved by using the following statements to call the OPTMILP procedure:

```
proc optmilp data=ex1data primalout=ex1soln;
run;
```

The progress of the solver is shown in Output 13.1.1.

Output 13.1.1 Simple Integer Linear Program PROC OPTMILP Log

```
NOTE: The problem ex1data has 10 variables (0 binary, 10 integer, 0 free, 0
      fixed).
NOTE: The problem has 2 constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 20 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 2 variables and 0 constraints.
NOTE: The MILP presolver removed 4 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 8 variables, 2 constraints, and 16 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       3     85.0000000    158.0000000   46.20%       0
             0       1       3     85.0000000     88.0955497    3.51%       0
             0       1       3     85.0000000     88.0955497    3.51%       0
             0       1       3     85.0000000     88.0955497    3.51%       0
NOTE: The MILP presolver is applied again.
             0       1       3     85.0000000     88.0955497    3.51%       0
             0       1       3     85.0000000     88.0626822    3.48%       0
             0       1       3     85.0000000     87.8820655    3.28%       0
             0       1       3     85.0000000     87.8539763    3.25%       0
             0       1       3     85.0000000     87.7208690    3.10%       0
             0       1       3     85.0000000     87.7180302    3.10%       0
             0       1       3     85.0000000     87.7133502    3.09%       0
             0       1       3     85.0000000     87.7128245    3.09%       0
NOTE: The MILP solver added 2 cuts with 8 cut coefficients at the root.
             3       1       4     86.0000000     87.3854749    1.59%       0
             4       0       5     87.0000000     87.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 87.
NOTE: The data set WORK.EX1SOLN has 10 observations and 8 variables.
```

The data set ex1soln is shown in Output 13.1.2.

Output 13.1.2 Simple Integer Linear Program Solution

Example 1 Solution Data

Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value
z	.RHS.	x[1]	I	1	0	4	0
z	.RHS.	x[2]	I	2	0	4	0
z	.RHS.	x[3]	I	3	0	4	0
z	.RHS.	x[4]	I	4	0	4	0
z	.RHS.	x[5]	I	5	0	4	0
z	.RHS.	x[6]	I	6	0	4	3
z	.RHS.	x[7]	I	7	0	4	1
z	.RHS.	x[8]	I	8	0	4	4
z	.RHS.	x[9]	I	9	0	4	0
z	.RHS.	x[10]	I	10	0	4	3

The optimal solution is $x_6 = 3, x_7 = 1, x_8 = 4$, and $x_{10} = 3$, with a total value of 87. From this solution, you can compute the total volume used, which is 988 ($\leq V = 1000$); the total weight used is 499 ($\leq W = 500$). The problem summary and solution summary are shown in Output 13.1.3.

Output 13.1.3 Simple Integer Linear Program Summary

The OPTMILP Procedure

Problem Summary	
Problem Name	ex1data
Objective Sense	Maximization
Objective Function	z
RHS	.RHS.
Number of Variables	10
Bounded Above	0
Bounded Below	0
Bounded Above and Below	10
Free	0
Fixed	0
Binary	0
Integer	10
Number of Constraints	2
LE (<=)	2
EQ (=)	0
GE (>=)	0
Range	0
Constraint Coefficients	20

Output 13.1.3 *continued*

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	z
Solution Status	Optimal
Objective Value	87
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	0
Best Bound	87
Nodes	5
Iterations	160
Presolve Time	0.00
Solution Time	0.01

Example 13.2: MIPLIB Benchmark Instance

The following example illustrates the conversion of a standard MPS-format file into an MPS-format SAS data set. The problem is re-solved several times, each time by using a different control option. For such a small example, it is necessary to disable cuts and heuristics in order to see the computational savings gained by using other options. For larger or more complex examples, the benefits of using the various control options are more pronounced.

The standard set of MILP benchmark cases is called MIPLIB (Bixby et al. 1998, Achterberg, Koch, and Martin 2003) and can be found at http://miplib.zib.de/. The following statement uses the %MPS2SASD macro to convert an example from MIPLIB to a SAS data set:

```
%mps2sasd(mpsfile="bell3a.mps", outdata=mpsdata);
```

The problem can then be solved using PROC OPTMILP on the data set created by the conversion:

```
proc optmilp data=mpsdata allcuts=none heuristics=none logfreq=10000;
run;
```

The resulting log is shown in Output 13.2.1.

Output 13.2.1 MIPLIB PROC OPTMILP Log

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 44 variables and 58 constraints.
NOTE: The MILP presolver removed 142 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 89 variables, 65 constraints, and 205
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       0              .         869515        .       0
           891     136       1         907340         873167    3.91%       0
          1814     128       3         898595         874502    2.76%       0
          1874     126       4         878651         874502    0.47%       0
          5124    1341       5         878430         875369    0.35%       0
         10000    1517       5         878430         876097    0.27%       0
         14778       0       5         878430         878430    0.00%       0
NOTE: Optimal.
NOTE: Objective = 878430.316.
```

Suppose you do not have a bound for the solution. If there is an objective value that, even if it is not optimal, satisfies your requirements, then you can save time by using the TARGET= option. The following PROC OPTMILP call solves the problem with a target value of 880,000:

```
proc optmilp data=mpsdata allcuts=none heuristics=none logfreq=5000
             target=880000;
run;
```

The relevant results from this run are displayed in Output 13.2.2. In this case, there is a decrease in CPU time, but the objective value has increased.

Output 13.2.2 MIPLIB PROC OPTMILP Log with TARGET= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 44 variables and 58 constraints.
NOTE: The MILP presolver removed 142 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 89 variables, 65 constraints, and 205
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       0              .         869515        .       0
           891     136       1         907340         873167    3.91%       0
          1814     128       3         898595         874502    2.76%       0
          1874     126       4         878651         874502    0.47%       0
NOTE: Target reached.
NOTE: Objective of the best integer solution found = 878651.068.
```

When the objective value of a solution is within a certain relative gap of the optimal objective value, the procedure stops. The acceptable relative gap can be changed using the RELOBJGAP= option, as demonstrated in the following example:

```
proc optmilp data=mpsdata allcuts=none heuristics=none relobjgap=0.01;
run;
```

The relevant results from this run are displayed in Output 13.2.3. In this case, since the specified RELOBJGAP= value is larger than the default value, the number of nodes and the CPU time have decreased from their values in the original run. Note that these savings are exchanged for an increase in the objective value of the solution.

Output 13.2.3 MIPLIB PROC OPTMILP Log with RELOBJGAP= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 44 variables and 58 constraints.
NOTE: The MILP presolver removed 142 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 89 variables, 65 constraints, and 205
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       0              .         869515        .       0
           100      82       0              .         872669        .       0
           200     159       0              .         872669        .       0
           300     235       0              .         872908        .       0
           400     314       0              .         872908        .       0
           500     392       0              .         872935        .       0
           600     463       0              .         873094        .       0
           700     534       0              .         873094        .       0
           800      46       0              .         873167        .       0
           891     136       1         907340         873167    3.91%       0
           900     144       1         907340         873167    3.91%       0
          1000      56       2         907340         873441    3.88%       0
          1100     132       2         907340         874032    3.81%       0
          1200     210       2         907340         874058    3.81%       0
          1300     289       2         907340         874165    3.80%       0
          1400     368       2         907340         874218    3.79%       0
          1500     443       2         907340         874314    3.78%       0
          1600     517       2         907340         874435    3.76%       0
          1700      17       2         907340         874502    3.76%       0
          1800     117       2         907340         874502    3.76%       0
          1814     128       3         898595         874502    2.76%       0
          1874     126       4         878651         874502    0.47%       0
NOTE: Optimal within relative gap.
NOTE: Objective = 878651.068.
```

The MAXTIME= option enables you to accept the best solution produced by PROC OPTMILP in a specified amount of time. The following example illustrates the use of the MAXTIME= option:

```
proc optmilp data=mpsdata allcuts=none heuristics=none maxtime=0.1;
run;
```

The relevant results from this run are displayed in Output 13.2.4. Once again, a reduction in solution time is traded for an increase in objective value.

Output 13.2.4 MIPLIB PROC OPTMILP Log with MAXTIME= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 44 variables and 58 constraints.
NOTE: The MILP presolver removed 142 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 89 variables, 65 constraints, and 205
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       0              .         869515        .       0
           100      82       0              .         872669        .       0
           200     159       0              .         872669        .       0
           300     235       0              .         872908        .       0
           400     314       0              .         872908        .       0
           500     392       0              .         872935        .       0
           600     463       0              .         873094        .       0
           700     534       0              .         873094        .       0
           800      46       0              .         873167        .       0
           891     136       1         907340         873167    3.91%       0
           900     144       1         907340         873167    3.91%       0
          1000      56       2         907340         873441    3.88%       0
          1100     132       2         907340         874032    3.81%       0
          1200     210       2         907340         874058    3.81%       0
NOTE: Real time limit reached.
NOTE: Objective of the best integer solution found = 907340.09.
```

The MAXNODES= option enables you to limit the number of nodes generated by PROC OPTMILP. The following example illustrates the use of the MAXNODES= option:

```
proc optmilp data=mpsdata allcuts=none heuristics=none maxnodes=1000;
run;
```

The relevant results from this run are displayed in Output 13.2.5. PROC OPTMILP displays the best objective value of all the solutions produced.

Output 13.2.5 MIPLIB PROC OPTMILP Log with MAXNODES= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 44 variables and 58 constraints.
NOTE: The MILP presolver removed 142 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 89 variables, 65 constraints, and 205
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       0              .         869515        .       0
           100      82       0              .         872669        .       0
           200     159       0              .         872669        .       0
           300     235       0              .         872908        .       0
           400     314       0              .         872908        .       0
           500     392       0              .         872935        .       0
           600     463       0              .         873094        .       0
           700     534       0              .         873094        .       0
           800      46       0              .         873167        .       0
           891     136       1         907340         873167    3.91%       0
           900     144       1         907340         873167    3.91%       0
           999      55       2         907340         873441    3.88%       0
NOTE: Node limit reached.
NOTE: Objective of the best integer solution found = 907340.09.
```

Example 13.3: Facility Location

This advanced example demonstrates how to warm start PROC OPTMILP by using the PRIMALIN= option. The model is constructed in PROC OPTMODEL and saved in an MPS-format SAS data set for use in PROC OPTMILP. This problem can also be solved from within PROC OPTMODEL; see Chapter 8 for details.

Consider the classical facility location problem. Given a set L of customer locations and a set F of candidate facility sites, you must decide on which sites to build facilities and assign coverage of customer demand to these sites so as to minimize cost. All customer demand d_i must be satisfied, and each facility has a demand capacity limit C. The total cost is the sum of the distances c_{ij} between facility j and its assigned customer i, plus a fixed charge f_j for building a facility at site j. Let $y_j = 1$ represent choosing site j to build a facility, and 0 otherwise. Also, let $x_{ij} = 1$ represent the assignment of customer i to facility j, and 0 otherwise. This model can be formulated as the following integer linear program:

$$
\begin{array}{lllll}
\min & \displaystyle\sum_{i\in L}\sum_{j\in F} c_{ij}x_{ij} + \sum_{j\in F} f_j y_j & & & \\
\text{s.t.} & \displaystyle\sum_{j\in F} x_{ij} = 1 & \forall i \in L & \text{(assign\_def)} \\
& x_{ij} \le y_j & \forall i \in L, j \in F & \text{(link)} \\
& \displaystyle\sum_{i\in L} d_i x_{ij} \le C y_j & \forall j \in F & \text{(capacity)} \\
& x_{ij} \in \{0,1\} & \forall i \in L, j \in F & \\
& y_j \in \{0,1\} & \forall j \in F &
\end{array}
$$

Constraint (assign_def) ensures that each customer is assigned to exactly one site. Constraint (link) forces a facility to be built if any customer has been assigned to that facility. Finally, constraint (capacity) enforces the capacity limit at each site.

Consider also a variation of this same problem where there is no cost for building a facility. This problem is typically easier to solve than the original problem. For this variant, let the objective be

$$\min \sum_{i\in L}\sum_{j\in F} c_{ij}x_{ij}$$

First, construct a random instance of this problem by using the following DATA steps:

```
%let NumCustomers  = 50;
%let NumSites      = 10;
%let SiteCapacity  = 35;
%let MaxDemand     = 10;
%let xmax          = 200;
%let ymax          = 100;
%let seed          = 938;

/* generate random customer locations */
data cdata(drop=i);
   length name $8;
   do i = 1 to &NumCustomers;
      name = compress('C'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
      demand = ranuni(&seed) * &MaxDemand;
      output;
   end;
run;

/* generate random site locations and fixed charge */
data sdata(drop=i);
   length name $8;
   do i = 1 to &NumSites;
      name = compress('SITE'||put(i,best.));
      x = ranuni(&seed) * &xmax;
```

```
        y = ranuni(&seed) * &ymax;
        fixed_charge = 30 * (abs(&xmax/2-x) + abs(&ymax/2-y));
        output;
     end;
  run;
```

The following PROC OPTMODEL statements generate the model and define both variants of the cost function:

```
proc optmodel;
   set <str> CUSTOMERS;
   set <str> SITES init {};
   /* x and y coordinates of CUSTOMERS and SITES */
   num x {CUSTOMERS union SITES};
   num y {CUSTOMERS union SITES};
   num demand {CUSTOMERS};
   num fixed_charge {SITES};
   /* distance from customer i to site j */
   num dist {i in CUSTOMERS, j in SITES}
       = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);
   read data cdata into CUSTOMERS=[name] x y demand;
   read data sdata into SITES=[name] x y fixed_charge;
   var Assign {CUSTOMERS, SITES} binary;
   var Build {SITES} binary;
   /* each customer assigned to exactly one site */
   con assign_def {i in CUSTOMERS}:
       sum {j in SITES} Assign[i,j] = 1;
   /* if customer i assigned to site j, then facility must be */
   /* built at j                                              */
   con link {i in CUSTOMERS, j in SITES}:
       Assign[i,j] <= Build[j];
   /* each site can handle at most &SiteCapacity demand */
   con capacity {j in SITES}:
       sum {i in CUSTOMERS} demand[i] * Assign[i,j]
       <= &SiteCapacity * Build[j];
   min CostNoFixedCharge
       = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j];
   save mps nofcdata;
   min CostFixedCharge
       = CostNoFixedCharge
         + sum {j in SITES} fixed_charge[j] * Build[j];
   save mps fcdata;
quit;
```

First solve the problem for the model with no fixed charge by using the following statements. The first PROC SQL call populates the macro variables varcostNo. This macro variable displays the objective value when the results are plotted. The second PROC SQL call generates a data set that is used to plot the results. The information printed in the log by PROC OPTMILP is displayed in Output 13.3.1.

```
proc optmilp data=nofcdata primalout=nofcout;
run;
proc sql noprint;
   select put(sum(_objcoef_ * _value_),6.1) into :varcostNo
   from nofcout;
quit;

proc sql;
   create table CostNoFixedCharge_Data as
   select
      scan(p._var_,2,'[],') as customer,
      scan(p._var_,3,'[],') as site,
      c.x as xi, c.y as yi, s.x as xj, s.y as yj
   from
      cdata as c,
      sdata as s,
      nofcout(where=(substr(_var_,1,6)='Assign' and
                     round(_value_) = 1)) as p
   where calculated customer = c.name and calculated site = s.name;
quit;
```

Output 13.3.1 PROC OPTMILP Log for Facility Location with No Fixed Charges

```
NOTE: The problem nofcdata has 510 variables (510 binary, 0 integer, 0 free, 0
      fixed).
NOTE: The problem has 560 constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 10 variables and 500 constraints.
NOTE: The MILP presolver removed 1010 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 500 variables, 60 constraints, and 1000
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       2    972.1737321              0    972.2       0
             0       1       2    972.1737321    961.2403449    1.14%       0
             0       1       2    972.1737321    961.2403449    1.14%       0
             0       1       2    972.1737321    961.2403449    1.14%       0
NOTE: The MILP presolver is applied again.
             0       1       4    966.4832160    962.9120804    0.37%       0
             0       1       5    966.4832160    966.4832160    0.00%       0
             0       0       5    966.4832160    966.4832160    0.00%       0
NOTE: The MILP solver added 0 cuts with 0 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 966.48321599.
NOTE: The data set WORK.NOFCOUT has 510 observations and 8 variables.
```

Next, solve the fixed-charge model by using the following statements. Note that the solution to the model with no fixed charge is feasible for the fixed-charge model and should provide a good starting point for PROC OPTMILP. The PRIMALIN= option provides an incumbent solution ("warm start"). The two PROC SQL calls perform the same functions as in the case with no fixed charges. The results from this approach are shown in Output 13.3.2.

```
proc optmilp data=fcdata primalin=nofcout primalout=fcout;
run;
proc sql noprint;
   select put(sum(_objcoef_ * _value_), 6.1) into :varcost
   from fcout(where=(substr(_var_,1,6)='Assign'));
   select put(sum(_objcoef_ * _value_), 5.1) into :fixcost
   from fcout(where=(substr(_var_,1,5)='Build'));
   select put(sum(_objcoef_ * _value_), 6.1) into :totalcost
   from fcout;
quit;
proc sql;
   create table CostFixedCharge_Data as
   select
      scan(p._var_,2,'[],') as customer,
      scan(p._var_,3,'[],') as site,
      c.x as xi, c.y as yi, s.x as xj, s.y as yj
   from
      cdata as c,
      sdata as s,
      fcout(where=(substr(_var_,1,6)='Assign' and
                   round(_value_) = 1)) as p
   where calculated customer = c.name and calculated site = s.name;
quit;
```

Output 13.3.2 PROC OPTMILP Log for Facility Location with Fixed Charges, Using Warm Start

```
NOTE: The problem fcdata has 510 variables (510 binary, 0 integer, 0 free, 0
      fixed).
NOTE: The problem has 560 constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 510 variables, 560 constraints, and 2010
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound       Gap    Time
             0       1       3  16070.0150023              0    16070       0
             0       1       3  16070.0150023   9946.2514269   61.57%       0
             0       1       3  16070.0150023   9962.4849932   61.31%       0
             0       1       3  16070.0150023   9971.5893075   61.16%       0
             0       1       3  16070.0150023   9974.5588580   61.11%       0
             0       1       3  16070.0150023   9978.1322942   61.05%       0
             0       1       3  16070.0150023   9978.3312183   61.05%       0
             0       1       3  16070.0150023   9980.4930282   61.01%       0
             0       1       3  16070.0150023   9981.2701907   61.00%       0
             0       1       4  16009.9864088   9981.2701907   60.40%       0
NOTE: The MILP solver added 20 cuts with 631 cut coefficients at the root.
            13       8       5  11115.0946280  10027.3840001   10.85%       0
            92      38       6  10986.9615361  10298.3588908    6.69%       0
           101      33       7  10966.7191607  10405.7274389    5.39%       0
           143      14       8  10959.4361909  10789.5675920    1.57%       0
           276      48       9  10950.0345545  10945.3793911    0.04%       0
           348      55      10  10948.4603465  10946.5600201    0.02%       0
           377      40      10  10948.4603465  10947.6598816    0.01%       0
NOTE: Optimal within relative gap.
NOTE: Objective = 10948.460346.
NOTE: The data set WORK.FCOUT has 510 observations and 8 variables.
```

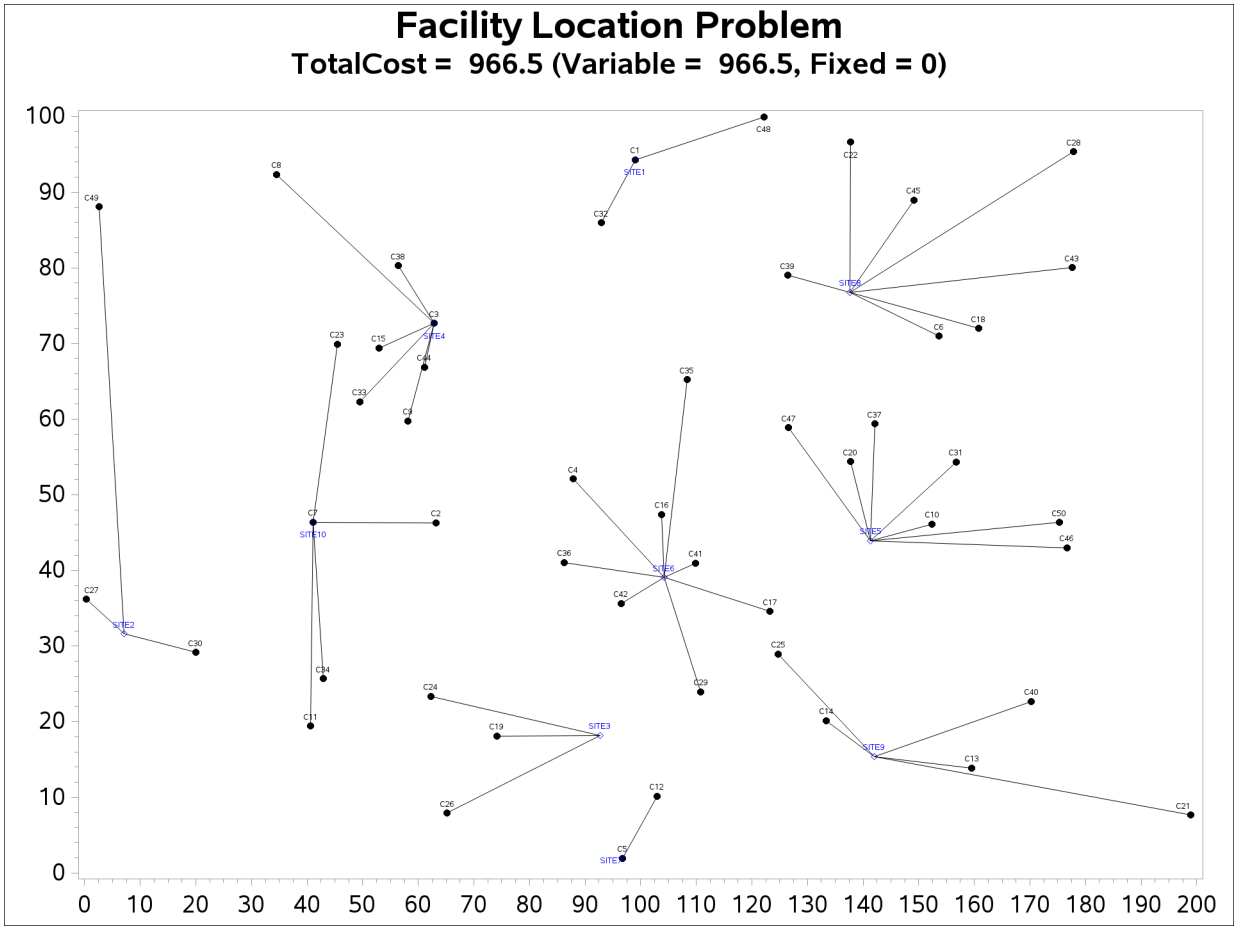
The following two SAS programs produce a plot of the solutions for both variants of the model, using data sets produced by PROC SQL from the PRIMALOUT= data sets produced by PROC OPTMILP.

NOTE: Execution of this code requires SAS/GRAPH software.

```
title1 "Facility Location Problem";
title2 "TotalCost = &varcostNo (Variable = &varcostNo, Fixed = 0)";
data csdata;
   set cdata(rename=(y=cy)) sdata(rename=(y=sy));
run;
/* create Annotate data set to draw line between customer and */
/* assigned site                                              */
%annomac;
data anno(drop=xi yi xj yj);
   %SYSTEM(2, 2, 2);
set CostNoFixedCharge_Data(keep=xi yi xj yj);
   %LINE(xi, yi, xj, yj, *, 1, 1);
run;
proc gplot data=csdata anno=anno;
   axis1 label=none order=(0 to &xmax by 10);
   axis2 label=none order=(0 to &ymax by 10);
   symbol1 value=dot interpol=none
      pointlabel=("#name" nodropcollisions height=0.7) cv=black;
   symbol2 value=diamond interpol=none
      pointlabel=("#name" nodropcollisions color=blue height=0.7) cv=blue;
   plot cy*x sy*x / overlay haxis=axis1 vaxis=axis2;
run;
quit;
```

The output from the first program appears in Output 13.3.3.

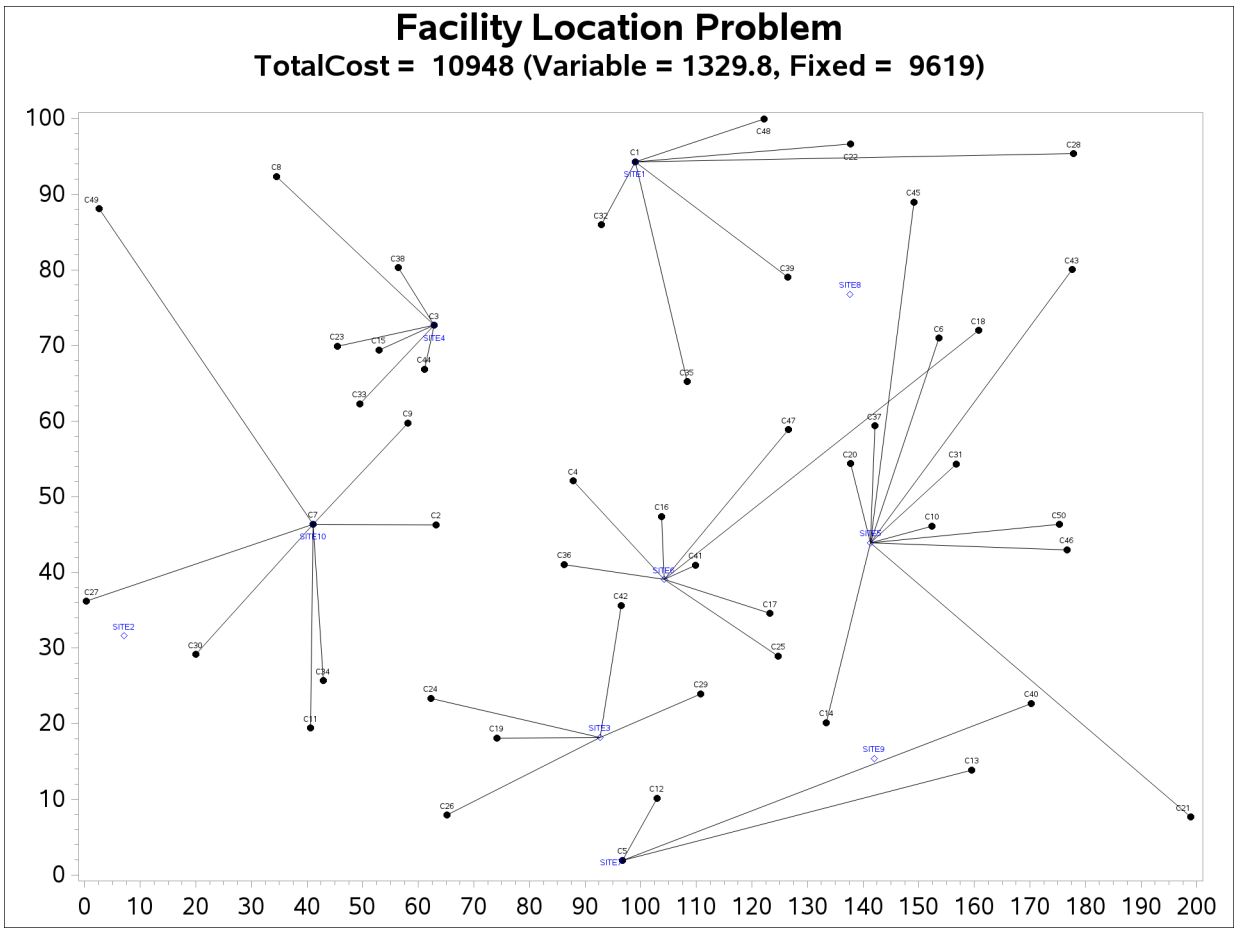
Output 13.3.3 Solution Plot for Facility Location with No Fixed Charges




```
title1 "Facility Location Problem";
title2 "TotalCost = &totalcost (Variable = &varcost, Fixed = &fixcost)";
/* create Annotate data set to draw line between customer and */
/* assigned site                                              */
data anno(drop=xi yi xj yj);
   %SYSTEM(2, 2, 2);
   set CostFixedCharge_Data(keep=xi yi xj yj);
   %LINE(xi, yi, xj, yj, *, 1, 1);
run;
proc gplot data=csdata anno=anno;
   axis1 label=none order=(0 to &xmax by 10);
   axis2 label=none order=(0 to &ymax by 10);
   symbol1 value=dot interpol=none
      pointlabel=("#name" nodropcollisions height=0.7) cv=black;
   symbol2 value=diamond interpol=none
      pointlabel=("#name" nodropcollisions color=blue height=0.7) cv=blue;
   plot cy*x sy*x / overlay haxis=axis1 vaxis=axis2;
run;
quit;
```

The output from the second program appears in Output 13.3.4.

Output 13.3.4 Solution Plot for Facility Location with Fixed Charges



The economic tradeoff for the fixed-charge model forces you to build fewer sites and push more demand to each site.

Example 13.4: Scheduling

This example is intended for users who prefer to use the SAS DATA step, PROC SQL, and similar programming methods to prepare data for input to SAS/OR optimization procedures. SAS/OR users who prefer to use the algebraic modeling capabilities of PROC OPTMODEL to specify optimization models should consult Example 8.1 in Chapter 8, "The Mixed Integer Linear Programming Solver," for a discussion of the same business problem in a PROC OPTMODEL context.

Scheduling is an application area where techniques in model generation can be valuable. Problems that involve scheduling are often solved with integer programming and are similar to assignment problems. In this example, you have eight one-hour time slots in each of five days. You have to assign four people to these time slots so that each slot is covered every day. You allow the people to specify preference data for each slot on each day. In addition, there are constraints that must be satisfied:

- Each person has some slots for which they are unavailable.

- Each person must have either slot 4 or 5 off for lunch.
- Each person can work only two time slots in a row.
- Each person can work only a specified number of hours in the week.

To formulate this problem, let i denote person, j denote time slot, and k denote day. Then, let $x_{ijk} = 1$ if person i is assigned to time slot j on day k, and 0 otherwise; let p_{ijk} denote the preference of person i for slot j on day k; and let h_i denote the number of hours in a week that person i will work. Then, you get

$$
\begin{array}{lll}
\max & \sum_{ijk} p_{ijk} x_{ijk} & \\
\text{subject to} & \sum_{i} x_{ijk} = 1 & \text{for all } j \text{ and } k \\
& x_{i4k} + x_{i5k} \leq 1 & \text{for all } i \text{ and } k \\
& x_{i,\ell,k} + x_{i,\ell+1,k} + x_{i,\ell+2,k} \leq 2 & \text{for all } i \text{ and } k, \text{ and } \ell = 1, \ldots, 6 \\
& \sum_{jk} x_{ijk} \leq h_i & \text{for all } i \\
& x_{ijk} = 0 \text{ or } 1 & \text{for all } i \text{ and } k \text{ such that } p_{ijk} > 0, \\
& & \quad \text{otherwise } x_{ijk} = 0
\end{array}
$$

To solve this problem, create a data set that has the hours and preference data for each individual, time slot, and day. A 10 represents the most desirable time slot, and a 1 represents the least desirable time slot. In addition, a 0 indicates that the time slot is not available.

```
data raw;
   input name $ hour slot mon tue wed thu fri;
   datalines;
marc  20   1    10 10 10 10 10
marc  20   2     9  9  9  9  9
marc  20   3     8  8  8  8  8
marc  20   4     1  1  1  1  1
marc  20   5     1  1  1  1  1
marc  20   6     1  1  1  1  1
marc  20   7     1  1  1  1  1
marc  20   8     1  1  1  1  1
mike  20   1    10  9  8  7  6
mike  20   2    10  9  8  7  6
mike  20   3    10  9  8  7  6
mike  20   4    10  3  3  3  3
mike  20   5     1  1  1  1  1
mike  20   6     1  2  3  4  5
mike  20   7     1  2  3  4  5
mike  20   8     1  2  3  4  5
bill  20   1    10 10 10 10 10
bill  20   2     9  9  9  9  9
bill  20   3     8  8  8  8  8
bill  20   4     0  0  0  0  0
bill  20   5     1  1  1  1  1
bill  20   6     1  1  1  1  1
bill  20   7     1  1  1  1  1
bill  20   8     1  1  1  1  1
bob   20   1    10  9  8  7  6
bob   20   2    10  9  8  7  6
bob   20   3    10  9  8  7  6
```

```
bob   20   4    10  3  3  3  3
bob   20   5     1  1  1  1  1
bob   20   6     1  2  3  4  5
bob   20   7     1  2  3  4  5
bob   20   8     1  2  3  4  5
;
```

These data are read by the following DATA step, and an integer program is built to solve the problem. The model is saved in the data set named MODEL, which is constructed in the following steps:

1. The objective function is built using the data saved in the RAW data set.
2. The constraints that ensure that no one works during a time slot during which they are unavailable are built.
3. The constraints that require a person to be working in each time slot are built.
4. The constraints that allow each person time for lunch are added.
5. The constraints that restrict people to only two consecutive hours are added.
6. The constraints that limit the time that any one person works in a week are added.
7. The constraints that allow a person to be assigned only to a time slot for which he is available are added.

The statements to build each of these constraints follow the formulation closely.

```
data model;
   array workweek{5} mon tue wed thu fri;
   array hours{4} hours1 hours2 hours3 hours4;
   retain hours1-hours4;

   set raw end=eof;

   length _row_ $ 8 _col_ $ 8 _type_ $ 8;
   keep _type_ _col_ _row_ _coef_;

   if      name='marc' then i=1;
   else if name='mike' then i=2;
   else if name='bill' then i=3;
   else if name='bob'  then i=4;

   hours{i}=hour;

   /* build the objective function */

   do k=1 to 5;
      _col_='x'||put(i,1.)||put(slot,1.)||put(k,1.);

      _row_='object';
      _coef_=workweek{k} * 1000;
      output;
   end;

   /* build the rest of the model */
```

```
/* cannot work during unavailable slots */
do k=1 to 5;
   if workweek{k}=0 then do;
      _row_='off'||put(i,1.)||put(slot,1.)||put(k,1.);
      _type_='eq';
      _col_='_RHS_';
      _coef_=0;
     output;
      _col_='x'||put(i,1.)||put(slot,1.)||put(k,1.);
     _coef_=1;
     _type_=' ';
     output;
   end;
end;

if eof then do;
   _coef_=.;
   _col_=' ';

   /* every hour 1 person working */
   do j=1 to 8;
      do k=1 to 5;
         _row_='work'||put(j,1.)||put(k,1.);
         _type_='eq';
         _col_='_RHS_';
         _coef_=1;
         output;
         _coef_=1;
         _type_=' ';
         do i=1 to 4;
            _col_='x'||put(i,1.)||put(j,1.)||put(k,1.);
            output;
         end;
      end;
   end;

   /* each person has a lunch */
   do i=1 to 4;
      do k=1 to 5;
         _row_='lunch'||put(i,1.)||put(k,1.);
         _type_='le';
         _col_='_RHS_';
         _coef_=1;
         output;
         _coef_=1;
         _type_=' ';
         _col_='x'||put(i,1.)||'4'||put(k,1.);
         output;
         _col_='x'||put(i,1.)||'5'||put(k,1.);
         output;
      end;
   end;
```

```
      /* work at most 2 slots in a row */
      do i=1 to 4;
         do k=1 to 5;
            do l=1 to 6;
            _row_='seq'||put(i,1.)||put(k,1.)||put(l,1.);
            _type_='le';
            _col_='_RHS_';
            _coef_=2;
             output;
            _coef_=1;
            _type_=' ';
               do j=0 to 2;
                  _col_='x'||put(i,1.)||put(l+j,1.)||put(k,1.);
                  output;
               end;
            end;
         end;
      end;

      /* work at most n hours in a week */
      do i=1 to 4;
         _row_='capacit'||put(i,1.);
         _type_='le';
         _col_='_RHS_';
         _coef_=hours{i};
         output;
         _coef_=1;
         _type_=' ';
         do j=1 to 8;
            do k=1 to 5;
               _col_='x'||put(i,1.)||put(j,1.)||put(k,1.);
               output;
            end;
         end;
      end;
   end;
run;
```

Next, this SAS data set is converted to an MPS-format SAS data set by establishing the structure of the MPS format and through very minor conversions of the data.

```
/* the following code transforms the above sparse data set */
/* into an MPS-format data set                              */

/* generate the header of the MPS-format data set */
data mps0;
   format field1 field2 field3 $10.;
   format field4 10.;
   format field5 $10.;
   format field6 10.;
   field1 = 'NAME';
   field2 = '       ';
   field3 = 'PROBLEM';
   field4 = .;
```

```
   field5 = '       ';
   field6 = .;
   output;
   field1 = 'ROWS';
   field3 = '';
   output;
   field1 = 'MAX';
   field2 = 'object';
   field3 = '';
   output;
run;

/* generate rows */
proc sql;
   create table mps1 as
      select _type_ as field1, _row_ as field2 from model
         where _row_ eq 'object' and _type_ ne '' union
      select 'E' as field1, _row_ as field2 from model
         where _type_ eq 'eq' union
      select 'L' as field1, _row_ as field2 from model
         where _type_ eq 'le' union
      select 'G' as field1, _row_ as field2 from model
         where _type_ eq 'ge';
quit;

/* indicate start of columns section and declare type of all */
/* variables as integer                                      */
data mps2;
   format field1 field2 field3  $10.;
   format field4 10.;
   format field5 $10.;
   format field6 10.;
   field1 = 'COLUMNS';
   field2 = '         ';
   field3 = '         ';
   field4 = .;
   field5 = '         ';
   field6 = .;
   output;
   field1 = '         ';
   field2 = '.MARK0';
   field3 = "'MARKER'";
   field4 = .;
   field5 = "'INTORG'";
   field6 = .;
   output;
run;

/* generate columns */
data mps3;
   set model;
   format field1 field2 field3 $10.;
   format field4 10.;
```

```
   format field5 $10.;
   format field6 10.;
   keep field1-field6;
   field1 = '      ';
   field2 = _col_;
   field3 = _row_;
   field4 = _coef_;
   field5 = '      ';
   field6 = .;
   if field2 ne '_RHS_'  then do;
      output;
   end;
run;

/* sort columns by variable names */
proc sort data=mps3;
   by field2;
run;

/* indicate the end of the columns section */
data mps4;
   format field1 field2 field3  $10.;
   format field4 10.;
   format field5 $10.;
   format field6 10.;
   field1 = '         ';
   field2 = '.MARK1';
   field3 = "'MARKER'";
   field4 = .;
   field5 = "'INTEND'";
   field6 = .;
   output;
run;

/* indicate the start of the RHS section */
data mps5;
   format field1 field2 field3  $10.;
   format field4 10.;
   format field5 $10.;
   format field6 10.;
   field1 = 'RHS';
run;

/* generate RHS entries */
data mps6;
   set model;
   format field1 field2 field3 $10.;
   format field4 10.;
   format field5 $10.;
   format field6 10.;
   keep field1-field6;
   field1 = '      ';
   field2 = _col_;
   field3 = _row_;
```

```
      field4 = _coef_;
      field5 = '       ';
      field6 = .;
      if field2 eq '_RHS_'  then do;
         output;
      end;
   run;

   /* denote the end of the MPS-format data set */
   data mps7;
      format field1 field2 field3  $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      field1 = 'ENDATA';
   run;

   /* merge all sections of the MPS-format data set */
   data mps;
      format field1 field2 field3 $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      set mps0 mps1 mps2 mps3 mps4 mps5 mps6 mps7;
   run;
```

The model is solved using the OPTMILP procedure. The option PRIMALOUT=SOLUTION causes PROC OPTMILP to save the primal solution in the data set named SOLUTION.

```
   /* solve the binary program */
   proc optmilp data=mps
      printlevel=0 loglevel=0
      primalout=solution maxtime=1000;
   run;
```

The following DATA step takes the solution data set SOLUTION and generates a report data set named REPORT. It restores the original interpretation (person, shift, day) of the variable names xijk so that a more meaningful report can be written. Then PROC TABULATE is used to display a schedule that shows how the eight time slots are covered for the week.

```
/* report the solution */
title 'Reported Solution';

data report;
   set solution;
   keep name slot mon tue wed thu fri;
   if substr(_var_,1,1)='x' then do;
      if _value_>0 then do;
         n=substr(_var_,2,1);
         slot=substr(_var_,3,1);
         d=substr(_var_,4,1);
         if      n='1' then name='marc';
         else if n='2' then name='mike';
         else if n='3' then name='bill';
         else               name='bob';
         if      d='1' then mon=1;
         else if d='2' then tue=1;
         else if d='3' then wed=1;
         else if d='4' then thu=1;
         else               fri=1;
         output;
      end;
   end;
run;

proc format;
   value xfmt 1='  xxx   ';
run;

proc tabulate data=report;
   class name slot;
   var mon--fri;
   table (slot * name), (mon tue wed thu fri)*sum=' '*f=xfmt.
          /misstext=' ';
run;
```

Output 13.4.1 from PROC TABULATE summarizes the schedule. Notice that the constraint that requires a person to be assigned to each possible time slot on each day is satisfied.

Output 13.4.1 A Scheduling Problem

Reported Solution

slot	name	mon	tue	wed	thu	fri
1	bill		xxx		xxx	
	marc			xxx		xxx
	mike	xxx				
2	bob	xxx				
	marc		xxx	xxx	xxx	xxx
3	bill			xxx	xxx	xxx
	bob	xxx	xxx			
4	mike	xxx	xxx	xxx	xxx	xxx
5	bob	xxx	xxx	xxx	xxx	xxx
6	bob					xxx
	marc	xxx				
	mike		xxx	xxx	xxx	
7	bob	xxx	xxx			
	mike			xxx	xxx	xxx
8	bob	xxx		xxx	xxx	
	mike		xxx			xxx

Recall that PROC OPTMILP puts a character string in the macro variable _OROPTMILP_ that describes the characteristics of the solution on termination. This string can be parsed using macro functions, and the information obtained can be used in report writing. The variable can be written to the log with the following command:

```
%put &_OROPTMILP_;
```

This command produces the output shown in Output 13.4.2.

Output 13.4.2 _OROPTMILP_ Macro Variable

```
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=211000 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0 BEST_BOUND=211000 NODES=1 ITERATIONS=64
PRESOLVE_TIME=0.00 SOLUTION_TIME=0.01
```

From this output you learn, for example, that at termination the solution is integer-optimal and has an objective value of 211,000.

References

Achterberg, T., Koch, T., and Martin, A. (2003). "MIPLIB 2003." http://miplib.zib.de/.

Achterberg, T., Koch, T., and Martin, A. (2005). "Branching Rules Revisited." *Operations Research Letters* 33:42–54.

Andersen, E. D., and Andersen, K. D. (1995). "Presolving in Linear Programming." *Mathematical Programming* 71:221–245.

Atamturk, A. (2004). "Sequence Independent Lifting for Mixed-Integer Programming." *Operations Research* 52:487–490.

Bixby, R. E., Ceria, S., McZeal, C. M., and Savelsbergh, M. W. P. (1998). "An Updated Mixed Integer Programming Library: MIPLIB 3.0." *Optima* 58:12–15.

Dantzig, G. B., Fulkerson, R., and Johnson, S. M. (1954). "Solution of a Large-Scale Traveling Salesman Problem." *Operations Research* 2:393–410.

Gondzio, J. (1997). "Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method." *INFORMS Journal on Computing* 9:73–91.

Land, A. H., and Doig, A. G. (1960). "An Automatic Method for Solving Discrete Programming Problems." *Econometrica* 28:497–520.

Linderoth, J. T., and Savelsbergh, M. W. P. (1998). "A Computational Study of Search Strategies for Mixed Integer Programming." *INFORMS Journal on Computing* 11:173–187.

Marchand, H., Martin, A., Weismantel, R., and Wolsey, L. (1999). "Cutting Planes in Integer and Mixed Integer Programming." DP 9953, CORE, Université Catholique de Louvain.

Ostrowski, J. (2008). *Symmetry in Integer Programming*. Ph.D. diss., Lehigh University.

Savelsbergh, M. W. P. (1994). "Preprocessing and Probing Techniques for Mixed Integer Programming Problems." *ORSA Journal on Computing* 6:445–454.

Chapter 14
The OPTQP Procedure

Contents

Overview: OPTQP Procedure **685**
Getting Started: OPTQP Procedure **687**
Syntax: OPTQP Procedure **691**
Functional Summary 692
PROC OPTQP Statement 692
PERFORMANCE Statement 695
Details: OPTQP Procedure **695**
Output Data Sets 695
Interior Point Algorithm: Overview 698
Parallel Processing 699
Iteration Log for the OPTQP Procedure 700
ODS Tables 700
Irreducible Infeasible Set 704
Macro Variable _OROPTQP_ 705
Examples: OPTQP Procedure **706**
Example 14.1: Linear Least Squares Problem 707
Example 14.2: Portfolio Optimization 709
Example 14.3: Portfolio Selection with Transactions 712
References **714**

Overview: OPTQP Procedure

The OPTQP procedure solves quadratic programs—problems with quadratic objective function and a collection of linear constraints, including lower or upper bounds (or both) on the decision variables.

Mathematically, a quadratic programming (QP) problem can be stated as follows:

$$\begin{aligned} \min \quad & \tfrac{1}{2}\,\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

where

$\mathbf{Q}$	$\in$	$\mathbb{R}^{n \times n}$	is the quadratic (also known as Hessian) matrix
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the constraints matrix
$\mathbf{x}$	$\in$	$\mathbb{R}^{n}$	is the vector of decision variables
$\mathbf{c}$	$\in$	$\mathbb{R}^{n}$	is the vector of linear objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^{m}$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^{n}$	is the vector of lower bounds on the decision variables
$\mathbf{u}$	$\in$	$\mathbb{R}^{n}$	is the vector of upper bounds on the decision variables

The quadratic matrix $\mathbf{Q}$ is assumed to be symmetric; that is,

$$q_{ij} = q_{ji}, \quad \forall i, j = 1, \ldots, n$$

Indeed, it is easy to show that even if $\mathbf{Q} \neq \mathbf{Q}^{\mathrm{T}}$, the simple modification

$$\tilde{\mathbf{Q}} = \frac{1}{2}(\mathbf{Q} + \mathbf{Q}^{\mathrm{T}})$$

produces an equivalent formulation $\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} \equiv \mathbf{x}^{\mathrm{T}}\tilde{\mathbf{Q}}\mathbf{x}$; hence symmetry is assumed. When you specify a quadratic matrix, it suffices to list only lower triangular coefficients.

In addition to being symmetric, $\mathbf{Q}$ is also required to be positive semidefinite,

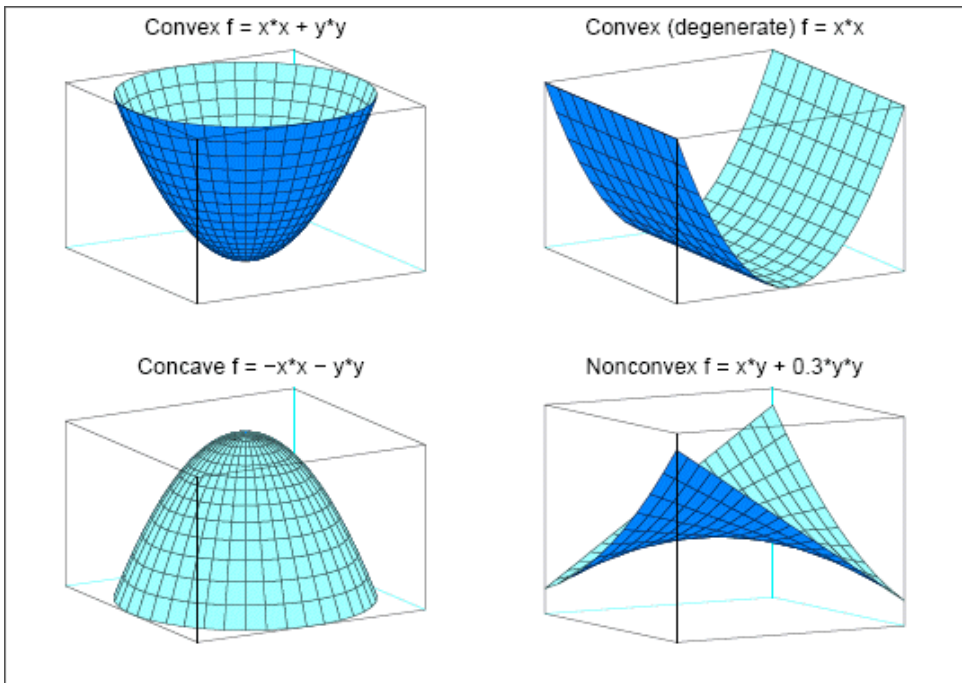
$$\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} \geq 0, \quad \forall \mathbf{x} \in \mathbb{R}^{n}$$

for minimization type of models; it is required to be negative semidefinite for the maximization type of models. Convexity can come as a result of a matrix-matrix multiplication

$$\mathbf{Q} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$$

or as a consequence of physical laws, and so on. See Figure 14.1 for examples of convex, concave, and nonconvex objective functions.

Figure 14.1 Examples of Convex, Concave, and Nonconvex Objective Functions



The order of constraints is insignificant. Some or all components of $\mathbf{l}$ or $\mathbf{u}$ (lower and upper bounds, respectively) can be omitted.

Getting Started: OPTQP Procedure

Consider a small illustrative example. Suppose you want to minimize a two-variable quadratic function $f(x_1, x_2)$ on the nonnegative quadrant, subject to two constraints:

$$\begin{array}{rrcrcrcrcl} \min & 2x_1 & + & 3x_2 & + & x_1^2 & + & 10x_2^2 & + & 2.5x_1x_2 \\ \text{subject to} & x_1 & - & x_2 & \leq & 1 & & & & \\ & x_1 & + & 2x_2 & \geq & 100 & & & & \\ & x_1 & & & \geq & 0 & & & & \\ & & & x_2 & \geq & 0 & & & & \end{array}$$

The linear objective function coefficients, vector of right-hand sides, and lower and upper bounds are identified immediately as

$$\mathbf{c} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 100 \end{bmatrix}, \quad \mathbf{l} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} +\infty \\ +\infty \end{bmatrix}$$

Carefully construct the quadratic matrix $\mathbf{Q}$. Observe that you can use symmetry to separate the main-diagonal and off-diagonal elements:

$$\frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} \equiv \frac{1}{2}\sum_{i,j=1}^{n} x_i\, q_{ij}\, x_j = \frac{1}{2}\sum_{i=1}^{n} q_{ii}\, x_i^2 + \sum_{i>j} x_i\, q_{ij}\, x_j$$

The first expression

$$\frac{1}{2}\sum_{i=1}^{n} q_{ii}\, x_i^2$$

sums the main-diagonal elements. Thus, in this case you have

$$q_{11} = 2, \quad q_{22} = 20$$

Notice that the main-diagonal values are doubled in order to accommodate the 1/2 factor. Now the second term

$$\sum_{i>j} x_i\, q_{ij}\, x_j$$

sums the off-diagonal elements in the strict lower triangular part of the matrix. The only off-diagonal ($x_i\, x_j,\ i \neq j$) term in the objective function is $2.5\, x_1\, x_2$, so you have

$$q_{21} = 2.5$$

Notice that you do not need to specify the upper triangular part of the quadratic matrix.

Finally, the matrix of constraints is as follows:

$$\mathbf{A} = \begin{bmatrix} 1 & -1 \\ 1 & 2 \end{bmatrix}$$

The SAS input data set with a quadratic programming system (QPS) format for the preceding problem can be expressed in the following manner:

```
data gsdata;
   input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
   datalines;
NAME      .       EXAMPLE    .            .          .
ROWS      .       .          .            .          .
N         OBJ     .          .            .          .
L         R1      .          .            .          .
G         R2      .          .            .          .
COLUMNS   .       .          .            .          .
.         X1      R1         1.0          R2         1.0
.         X1      OBJ        2.0          .          .
.         X2      R1        -1.0          R2         2.0
.         X2      OBJ        3.0          .          .
RHS       .       .          .            .          .
.         RHS     R1         1.0          .          .
.         RHS     R2         100          .          .
RANGES    .       .          .            .          .
BOUNDS    .       .          .            .          .
QUADOBJ   .       .          .            .          .
.         X1      X1         2.0          .          .
.         X1      X2         2.5          .          .
.         X2      X2         20           .          .
ENDATA    .       .          .            .          .
;
```

For more details about the QPS-format data set, see Chapter 17, "The MPS-Format SAS Data Set."

Alternatively, if you have a QPS-format flat file named gs.qps, then the following call to the SAS macro %MPS2SASD translates that file into a SAS data set, named gsdata:

```
%mps2sasd(mpsfile =gs.qps, outdata = gsdata);
```

NOTE: The SAS macro %MPS2SASD is provided in SAS/OR software. See "Converting an MPS/QPS-Format File: %MPS2SASD" on page 848 for details.

You can use the following call to PROC OPTQP:

```
proc optqp data=gsdata
   primalout = gspout
   dualout   = gsdout;
run;
```

The procedure output is displayed in Figure 14.2.

Figure 14.2 Procedure Output

The OPTQP Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Problem Summary	
Problem Name	EXAMPLE
Objective Sense	Minimization
Objective Function	OBJ
RHS	RHS
Number of Variables	2
Bounded Above	0
Bounded Below	2
Bounded Above and Below	0
Free	0
Fixed	0
Number of Constraints	2
LE (<=)	1
EQ (=)	0
GE (>=)	1
Range	0
Constraint Coefficients	4
Hessian Diagonal Elements	2
Hessian Elements Above the Diagonal	1

Figure 14.2 *continued*

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	OBJ
Solution Status	Optimal
Objective Value	15018
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Duality Gap	3.633377E-16
Complementarity	0
Iterations	6
Presolve Time	0.00
Solution Time	0.39

The optimal primal solution is displayed in Figure 14.3.

Figure 14.3 Optimal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Linear Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status
1	OBJ	RHS	X1	N	2	0	1.7977E308	34	O
2	OBJ	RHS	X2	N	3	0	1.7977E308	33	O

The SAS log shown in Figure 14.4 provides information about the problem, convergence information after each iteration, and the optimal objective value.

Figure 14.4 Iteration Log

```
NOTE: The problem EXAMPLE has 2 variables (0 free, 0 fixed).
NOTE: The problem has 2 constraints (1 LE, 0 EQ, 1 GE, 0 range).
NOTE: The problem has 4 constraint coefficients.
NOTE: The objective function has 2 Hessian diagonal elements and 1 Hessian
      elements above the diagonal.
NOTE: The QP presolver value AUTOMATIC is applied.
NOTE: The QP presolver removed 0 variables and 0 constraints.
NOTE: The QP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 2 variables, 2 constraints, and 4 constraint
      coefficients.
NOTE: The QP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Interior Point algorithm is using up to 4 threads.
                                        Primal       Bound        Dual
      Iter  Complement Duality Gap      Infeas       Infeas       Infeas   Time
         0  3.5863E+03  4.8823E+00  1.0251E+00  1.0354E+02  2.3142E-15      0
         1  1.9345E+03  9.6222E-01  4.4158E-01  4.4602E+01  5.7855E-16      0
         2  2.2140E+03  1.2297E-01  4.4158E-03  4.4602E-01  5.5926E-15      0
         3  5.0020E+01  3.2272E-03  4.4158E-05  4.4602E-03  9.6502E-15      0
         4  4.9973E-01  3.2332E-05  4.4158E-07  4.4602E-05  2.5148E-14      0
         5  4.9972E-03  3.2332E-07  4.4158E-09  4.4602E-07  3.0701E-14      0
         6  0.0000E+00  3.6334E-16  1.5730E-16  0.0000E+00  1.2342E-14      0
NOTE: Optimal.
NOTE: Objective = 15018.
NOTE: The Interior Point solve time is 0.00 seconds.
NOTE: The data set WORK.GSPOUT has 2 observations and 9 variables.
NOTE: The data set WORK.GSDOUT has 2 observations and 10 variables.
```

See the section "Interior Point Algorithm: Overview" on page 698 and the section "Iteration Log for the OPTQP Procedure" on page 700 for more details about convergence information given by the iteration log.

Syntax: OPTQP Procedure

The following statements are available in the OPTQP procedure:

PROC OPTQP < *options* > **;**
PERFORMANCE < *performance-options* > **;**

Functional Summary

Table 14.1 outlines the options available for the OPTQP procedure classified by function.

Table 14.1 Options in the OPTQP Procedure

Description	Option
Data Set Options	
Specifies a QPS-format input SAS data set	DATA=
Specifies a dual solution output SAS data set	DUALOUT=
Specifies whether the QP model is a maximization or minimization problem	OBJSENSE=
Specifies the primal solution output SAS data set	PRIMALOUT=
Saves output data sets only if optimal	SAVE_ONLY_IF_OPTIMAL
Solver Options	
Enables or disables IIS detection	IIS=
Control Options	
Specifies the maximum number of iterations	MAXITER=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the type of presolve	PRESOLVER=
Enables or disables iteration log	LOGFREQ=
Enables or disables printing summary	PRINTLEVEL=
Specifies the stopping criterion based on duality gap	STOP_DG=
Specifies the stopping criterion based on dual infeasibility	STOP_DI=
Specifies the stopping criterion based on primal infeasibility	STOP_PI=
Specifies units of CPU time or real time	TIMETYPE=

PROC OPTQP Statement

The following options can be specified in the PROC OPTQP statement.

DATA=*SAS-data-set*

specifies the input SAS data set. This data set can also be created from a QPS-format flat file by using the SAS macro %MPS2SASD. If the DATA= option is not specified, PROC OPTQP uses the most recently created SAS data set. See Chapter 17, "The MPS-Format SAS Data Set," for more details.

DUALOUT=*SAS-data-set*

DOUT=*SAS-data-set*

specifies the output data set to contain the dual solution. See the section "Output Data Sets" on page 695 for details.

IIS=*number* | *string*

specifies whether PROC OPTQP attempts to identify a set of constraints and variables that form an irreducible infeasible set (IIS). Table 14.2 describes the valid values of the IIS= option.

Table 14.2 Values for IIS= Option

number	*string*	**Description**
0	OFF	Disables IIS detection.
1	ON	Enables IIS detection.

If an IIS is found, you can find information about infeasible constraints or variable bounds in the DUALOUT= and PRIMALOUT= data sets. The default value of this option is OFF. See the section "Irreducible Infeasible Set" on page 704 for details.

LOGFREQ=*k*

PRINTFREQ=*k*

specifies that the printing of the solution progress to the iteration log should occur after every *k* iterations. The print frequency, *k*, is an integer between zero and the largest four-byte, signed integer, which is $2^{31} - 1$. The value $k = 0$ disables the printing of the progress of the solution. The default value of this option is 1.

MAXITER=*k*

specifies the maximum number of predictor-corrector iterations performed by the interior point algorithm (see the section "Interior Point Algorithm: Overview" on page 698). The value *k* is an integer between 1 and the largest four-byte, signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed.

MAXTIME=*t*

specifies an upper limit of *t* seconds of time for reading in the data and performing the optimization process. The value of the TIMETYPE= option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OBJSENSE=*option*

specifies whether the QP model is a minimization or a maximization problem. You specify OBJSENSE=MIN for a minimization problem and OBJSENSE=MAX for a maximization problem. Alternatively, you can specify the objective sense in the input data set; see the section "ROWS Section" on page 841 for details. If the objective sense is specified differently in these two places, this option supersedes the objective sense specified in the input data set. If the objective sense is not specified anywhere, then PROC OPTQP interprets and solves the quadratic program as a minimization problem.

PRESOLVER=*number* | *string*

PRESOL=*number* | *string*

specifies one of the following presolve options:

number	*string*	**Description**
0	NONE	Disables the presolver.
–1	AUTOMATIC	Applies the presolver by using default setting.

number	*string*	**Description**
1	BASIC	Applies the basic presolver.
2	MODERATE	Applies the moderate presolver.
3	AGGRESSIVE	Applies the aggressive presolver.

You can specify the option either by a word or by integers from –1 to 3. The default option is AUTOMATIC.

PRIMALOUT=*SAS-data-set*

POUT=*SAS-data-set*

specifies the output data set to contain the primal solution. See the section "Output Data Sets" on page 695 for details.

PRINTLEVEL=0 | 1 | 2

specifies whether a summary of the problem and solution should be printed. If PRINTLEVEL=1, then the Output Delivery System (ODS) tables ProblemSummary, SolutionSummary, and PerformanceInfo are produced and printed. If PRINTLEVEL=2, then the same tables are produced and printed along with an additional table called ProblemStatistics. If PRINTLEVEL=0, then no ODS tables are produced or printed. The default value is 1.

For details about the ODS tables created by PROC OPTQP, see the section "ODS Tables" on page 700.

SAVE_ONLY_IF_OPTIMAL

specifies that the PRIMALOUT= and DUALOUT= data sets be saved only if the final solution obtained by the solver at termination is optimal. If the PRIMALOUT= or DUALOUT= option is specified, and this option is not specified, then the output data sets will only contain solution values at optimality. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the output data sets will not contain an intermediate solution.

STOP_DG=δ

specifies the desired relative duality gap, $\delta \in$[1E–9, 1E–4]. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 698 for details.

STOP_DI=β

specifies the maximum allowed relative dual constraints violation, $\beta \in$[1E–9, 1E–4]. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 698 for details.

STOP_PI=α

specifies the maximum allowed relative bound and primal constraints violation, $\alpha \in$[1E–9, 1E–4]. The default value is 1E–6. See the section "Interior Point Algorithm: Overview" on page 698 for details.

TIMETYPE=*number* | *string*

specifies whether CPU time or real time is used for the MAXTIME= option and the _OROPTQP_ macro variable in a PROC OPTQP call. Table 14.4 describes the valid values of the TIMETYPE= option.

Table 14.4 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time.
1	REAL	Specifies units of real time.

The default value of the TIMETYPE= option depends on the value of the NTHREADS= option in the PERFORMANCE statement. See the section "PERFORMANCE Statement" on page 21 for more information about the NTHREADS= option.

If you specify a value greater than 1 for the NTHREADS= option, the default value of the TIMETYPE= option is REAL. If you specify a value of 1 for the NTHREADS= option, the default value of the TIMETYPE= option is CPU.

PERFORMANCE Statement

PERFORMANCE < *performance-options* > **;**

The PERFORMANCE statement specifies *performance-options* for multithreaded (SMP) computing, passes variables around the distributed computing environment, and requests detailed results about the performance characteristics of the OPTQP procedure.

The PERFORMANCE statement for multithreaded computing mode is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics." The OPTQP procedure supports the deterministic and nondeterministic modes of the PARALLELMODE= option in the PERFORMANCE statement.

Details: OPTQP Procedure

Output Data Sets

This section describes the PRIMALOUT= and DUALOUT= output data sets. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the output data sets do not contain an intermediate solution.

Definitions of Variables in the PRIMALOUT= Data Set

The PRIMALOUT= data set contains the primal solution to the quadratic programming (QP) model. The variables in the data set have the following names and meanings.

_OBJ_ID_
specifies the name of the objective function. Naming objective functions is particularly useful when there are multiple objective functions, in which case each objective function has a unique name. See the section "ROWS Section" on page 841 for details.

NOTE: PROC OPTQP does not support simultaneous optimization of multiple objective functions in this release.

_RHS_ID_
specifies the name of the variable that contains the right-hand-side value of each constraint. See the section "RHS Section (Optional)" on page 843 for details.

VAR
specifies the name of the decision variable.

TYPE
specifies the type of the decision variable. _TYPE_ can take one of the following values:

N nonnegative variable

D bounded variable with both finite lower and finite upper bound

F free variable

X fixed variable

O other

OBJCOEF
specifies the coefficient of the decision variable in the linear component of the objective function.

LBOUND
specifies the lower bound on the decision variable.

UBOUND
specifies the upper bound on the decision variable.

VALUE
specifies the value of the decision variable.

STATUS
specifies the status of the decision variable. _STATUS_ can indicate one of the following two cases:

O The QP problem is optimal.

I The QP problem could be infeasible or unbounded, or PROC OPTQP was not able to solve the problem.

The following values can appear only if IIS=ON. See the section "Irreducible Infeasible Set" on page 704 for details.

I_L The lower bound of the variable is needed for the IIS.

I_U The upper bound of the variable is needed for the IIS.

I_F Both bounds of the variable are needed for the IIS (the variable is fixed or has conflicting bounds).

Definitions of Variables in the DUALOUT= Data Set

The DUALOUT= data set contains the dual solution to the QP model. Information about the objective rows of the QP problems is not included. The variables in the data set have the following names and meanings.

_OBJ_ID_
specifies the name of the objective function. Naming objective functions is particularly useful when there are multiple objective functions, in which case each objective function has a unique name. See the section "ROWS Section" on page 841 for details.

NOTE: PROC OPTQP does not support simultaneous optimization of multiple objective functions in this release.

_RHS_ID_
specifies the name of the variable that contains the right-hand-side value of each constraint. See the section "RHS Section (Optional)" on page 843 for details.

ROW
specifies the name of the constraint. See the section "ROWS Section" on page 841 for details.

TYPE
specifies the type of the constraint. _TYPE_ can take one of the following values:

L "less than or equals" constraint

E equality constraint

G "greater than or equals" constraint

R ranged constraint (both "less than or equals" and "greater than or equals")

See the sections "ROWS Section" on page 841 and "RANGES Section (Optional)" on page 844 for details.

RHS
specifies the value of the right-hand side of the constraint. It takes a missing value for a ranged constraint.

_L_RHS_
specifies the lower bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_U_RHS_
specifies the upper bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

VALUE
specifies the value of the dual variable associated with the constraint.

STATUS
specifies the status of the constraint. _STATUS_ can indicate one of the following two cases:

O The QP problem is optimal.

I The QP problem could be infeasible or unbounded, or PROC OPTQP was not able to solve the problem.

The following values can appear only if option IIS=ON. See the section "Irreducible Infeasible Set" on page 704 for details.

I_L The "GE" ($\geq$) condition of the constraint is needed for the IIS.

I_U The "LE" ($\leq$) condition of the constraint is needed for the IIS.

I_F Both conditions of the constraint are needed for the IIS (the constraint is an equality or a range constraint with conflicting bounds).

ACTIVITY
specifies the value of a constraint. In other words, the value of _ACTIVITY_ for the *i*th constraint is equal to $\mathbf{a}_i^{\mathrm{T}}\mathbf{x}$, where $\mathbf{a}_i$ refers to the *i*th row of the constraints matrix and $\mathbf{x}$ denotes the vector of current decision variable values.

Interior Point Algorithm: Overview

The interior point solver in PROC OPTQP implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following QP formulation (the primal):

$$\begin{aligned} \min \quad & \tfrac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

The corresponding dual is as follows:

$$\begin{array}{rrcrcrcl} \max & -\frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} & + & \mathbf{b}^{\mathrm{T}}\mathbf{y} & & & & \\ \text{subject to} & -\mathbf{Q}\mathbf{x} & + & \mathbf{A}^{\mathrm{T}}\mathbf{y} & + & \mathbf{w} & = & \mathbf{c} \\ & & & & & \mathbf{y} & \geq & \mathbf{0} \\ & & & & & \mathbf{w} & \geq & \mathbf{0} \end{array}$$

where $\mathbf{y} \in \mathbb{R}^m$ refers to the vector of dual variables and $\mathbf{w} \in \mathbb{R}^n$ refers to the vector of slack variables in the dual problem.

The dual makes an important contribution to the certificate of optimality for the primal. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows:

$$\begin{aligned} \mathbf{A}\mathbf{x} - \mathbf{s} &= \mathbf{b} \quad \text{(primal feasibility)} \\ -\mathbf{Q}\mathbf{x} + \mathbf{A}^{\mathrm{T}}\mathbf{y} + \mathbf{w} &= \mathbf{c} \quad \text{(dual feasibility)} \\ \mathbf{W}\mathbf{X}\mathbf{e} &= \mathbf{0} \quad \text{(complementarity)} \\ \mathbf{S}\mathbf{Y}\mathbf{e} &= \mathbf{0} \quad \text{(complementarity)} \\ \mathbf{x},\, \mathbf{y},\, \mathbf{w},\, \mathbf{s} &\geq \mathbf{0} \end{aligned}$$

where $\mathbf{e} \equiv (1, \ldots, 1)^{\mathrm{T}}$ is of appropriate dimension and $\mathbf{s} \in \mathbb{R}^m$ is the vector of primal slack variables.

NOTE: Slack variables (the *s* vector) are automatically introduced by the solver when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the solver to handle slack variables much more efficiently.

The letters $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{W}$, and $\mathbf{S}$ denote matrices with corresponding *x*, *y*, *w*, and *s* on the main diagonal and zero elsewhere, as in the following example:

$$\mathbf{X} \equiv \begin{bmatrix} x_1 & 0 & \cdots & 0 \\ 0 & x_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x_n \end{bmatrix}$$

If $(\mathbf{x}^*, \mathbf{y}^*, \mathbf{w}^*, \mathbf{s}^*)$ is a solution of the previously defined system of equations that represent the KKT conditions, then $\mathbf{x}^*$ is also an optimal solution to the original QP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations as follows:

$$\begin{bmatrix} \mathbf{Y}^{-1}\mathbf{S} & \mathbf{A} \\ \mathbf{A}^{\mathrm{T}} & -\mathbf{Q} - \mathbf{X}^{-1}\mathbf{W} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{y} \\ \Delta\mathbf{x} \end{bmatrix} = \begin{bmatrix} \Xi \\ \Theta \end{bmatrix}$$

where $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ denote the vector of *search directions* in the primal and dual spaces, respectively, and Θ and Ξ constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. PROC OPTQP uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point solver is that it takes full advantage of the sparsity in the constraint and quadratic matrices, thereby enabling it to efficiently solve large-scale quadratic programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore, it is of interest to observe the following four measures where $\|v\|_2$ is the Euclidean norm of the vector v:

- relative primal infeasibility measure α:

$$\alpha = \frac{\|\mathbf{Ax} - \mathbf{b} - \mathbf{s}\|_2}{\|\mathbf{b}\|_2 + 1}$$

- relative dual infeasibility measure β:

$$\beta = \frac{\|\mathbf{Qx} + \mathbf{c} - \mathbf{A}^{\mathrm{T}}\mathbf{y} - \mathbf{w}\|_2}{\|\mathbf{c}\|_2 + 1}$$

- relative duality gap δ:

$$\delta = \frac{|\mathbf{x}^{\mathrm{T}}\mathbf{Qx} + \mathbf{c}^{\mathrm{T}}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{y}|}{|\frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Qx} + \mathbf{c}^{\mathrm{T}}\mathbf{x}| + 1}$$

- absolute complementarity γ:

$$\gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i$$

These measures are displayed in the iteration log.

Parallel Processing

The interior point algorithm can be run in single-machine mode (in single-machine mode, the computation is executed by multiple threads on a single computer). You can specify options that control parallel processing in the PERFORMANCE statement, which is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Iteration Log for the OPTQP Procedure

The interior point solver in PROC OPTQP implements an infeasible primal-dual predictor-corrector interior point algorithm. The following information is displayed in the iteration log:

Iter	indicates the iteration number.
Complement	indicates the (absolute) complementarity.
Duality Gap	indicates the (relative) duality gap.
Primal Infeas	indicates the (relative) primal infeasibility measure.
Bound Infeas	indicates the (relative) bound infeasibility measure.
Dual Infeas	indicates the (relative) dual infeasibility measure.
Time	indicates the time elapsed (in seconds).

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. Nonconvergence can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value that you specify in the MAXITER= or MAXTIME= option. If the complementarity or the duality gap does not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

ODS Tables

PROC OPTQP creates three Output Delivery System (ODS) tables by default. The first table, ProblemSummary, is a summary of the input QP problem. The second table, SolutionSummary, is a brief summary of the solution status. The third table, PerformanceInfo, is a summary of performance options. You can use ODS table names to select tables and create output data sets. For more information about ODS, see the *SAS Output Delivery System: User's Guide*.

If you specify a value of 2 for the PRINTLEVEL= option, then the ProblemStatistics table is produced. This table contains information about the problem data. See the section "Problem Statistics" on page 703 for more information.

If you specify the DETAILS option in the PERFORMANCE statement, then the Timing table is produced.

Table 14.5 lists all the ODS tables that can be produced by the OPTQP procedure, along with the statement and option specifications required to produce each table.

Table 14.5 ODS Tables Produced by PROC OPTQP

ODS Table Name	Description	Statement	Option
ProblemSummary	Summary of the input QP problem	PROC OPTQP	PRINTLEVEL=1 (default)
SolutionSummary	Summary of the solution status	PROC OPTQP	PRINTLEVEL=1 (default)
ProblemStatistics	Description of input problem data	PROC OPTQP	PRINTLEVEL=2
PerformanceInfo	List of performance options and their values	PROC OPTQP	PRINTLEVEL=1 (default)
Timing	Detailed solution timing	PERFORMANCE	DETAILS

A typical output of PROC OPTQP is shown in Output 14.5.

Figure 14.5 Typical OPTQP Output

The OPTQP Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Problem Summary	
Problem Name	BANDM
Objective Sense	Minimization
Objective Function	1
RHS	ZZZZ0001
Number of Variables	472
Bounded Above	0
Bounded Below	472
Bounded Above and Below	0
Free	0
Fixed	0
Number of Constraints	305
LE (<=)	0
EQ (=)	305
GE (>=)	0
Range	0
Constraint Coefficients	2494
Hessian Diagonal Elements	25
Hessian Elements Above the Diagonal	16

Solution Summary	
Solver	QP
Algorithm	Interior Point
Objective Function	1
Solution Status	Optimal
Objective Value	16352.342037
Primal Infeasibility	1.270665E-11
Dual Infeasibility	3.556547E-16
Bound Infeasibility	0
Duality Gap	9.470938E-12
Complementarity	1.1778485E-8
Iterations	22
Presolve Time	0.00
Solution Time	0.23

You can create output data sets from these tables by using the ODS OUTPUT statement. This can be useful, for example, when you want to create a report to summarize multiple PROC OPTQP runs. The output data sets that correspond to the preceding output are shown in Output 14.6, where you can also find (in the row following the heading of each data set in the display) the variable names that are used in the table definition (template) of each table.

Figure 14.6 ODS Output Data Sets

Problem Summary

Obs	Label1	cValue1	nValue1
1	Problem Name	BANDM	.
2	Objective Sense	Minimization	.
3	Objective Function	1	.
4	RHS	ZZZZ0001	.
5			.
6	Number of Variables	472	472.000000
7	Bounded Above	0	0
8	Bounded Below	472	472.000000
9	Bounded Above and Below	0	0
10	Free	0	0
11	Fixed	0	0
12			.
13	Number of Constraints	305	305.000000
14	LE (<=)	0	0
15	EQ (=)	305	305.000000
16	GE (>=)	0	0
17	Range	0	0
18			.
19	Constraint Coefficients	2494	2494.000000
20			.
21	Hessian Diagonal Elements	25	25.000000
22	Hessian Elements Above the Diagonal	16	16.000000

Figure 14.6 *continued*

Solution Summary

Obs	Label1	cValue1	nValue1
1	Solver	QP	.
2	Algorithm	Interior Point	.
3	Objective Function	1	.
4	Solution Status	Optimal	.
5	Objective Value	16352.342037	16352
6			.
7	Primal Infeasibility	1.270665E-11	1.270665E-11
8	Dual Infeasibility	3.556547E-16	3.556547E-16
9	Bound Infeasibility	0	0
10	Duality Gap	9.470938E-12	9.470938E-12
11	Complementarity	1.1778485E-8	1.1778485E-8
12			.
13	Iterations	22	22.000000
14	Presolve Time	0.00	0
15	Solution Time	0.23	0.234001

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTQP procedure causes the ODS table ProblemStatistics to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Output 14.7 demonstrates the contents of the ODS table ProblemStatistics.

Figure 14.7 ODS Table ProblemStatistics

The OPTQP Procedure

Problem Statistics	
Number of Constraint Matrix Nonzeros	4
Maximum Constraint Matrix Coefficient	2
Minimum Constraint Matrix Coefficient	1
Average Constraint Matrix Coefficient	1.25
Number of Linear Objective Nonzeros	2
Maximum Linear Objective Coefficient	3
Minimum Linear Objective Coefficient	2
Average Linear Objective Coefficient	2.5
Number of Lower Triangular Hessian Nonzeros	1
Number of Diagonal Hessian Nonzeros	2
Maximum Hessian Coefficient	20
Minimum Hessian Coefficient	2
Average Hessian Coefficient	6.75
Number of RHS Nonzeros	2
Maximum RHS	100
Minimum RHS	1
Average RHS	50.5
Maximum Number of Nonzeros per Column	2
Minimum Number of Nonzeros per Column	2
Average Number of Nonzeros per Column	2
Maximum Number of Nonzeros per Row	2
Minimum Number of Nonzeros per Row	2
Average Number of Nonzeros per Row	2

Irreducible Infeasible Set

For a quadratic programming problem, an irreducible infeasible set (IIS) is an infeasible subset of constraints and variable bounds that becomes feasible if any single constraint or variable bound is removed. It is possible to have more than one IIS in an infeasible QP. Identifying an IIS can help isolate the structural infeasibility in a QP. The IIS=ON option directs the OPTQP procedure to search for an IIS in a specified QP.

Whether a quadratic programming problem is feasible or infeasible is determined by its constraints and variable bounds, which have nothing to do with its objective function. When you specify the IIS=ON option, the OPTQP procedure treats this problem as a linear programming problem by ignoring its objective function. Then finding IIS is the same as what PROC OPTLP does with the IIS=ON option. See the section "Irreducible Infeasible Set" on page 597 in Chapter 12, "The OPTLP Procedure," for more information about the irreducible infeasible set.

Macro Variable _OROPTQP_

The OPTQP procedure defines a macro variable named _OROPTQP_. This variable contains a character string that indicates the status of the procedure. The various terms of the variable are interpreted as follows.

STATUS
indicates the solver status at termination. It can take one of the following values:

OK	The procedure terminated normally.
SYNTAX_ERROR	Incorrect syntax was used.
DATA_ERROR	The input data were inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem occurred in reading or writing data.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM
indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take the following value:

IP	The interior point algorithm produced the solution data.

SOLUTION_STATUS
indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
CONDITIONAL_OPTIMAL	The solution is optimal, but some infeasibilities (primal, dual or bound) exceed tolerances due to scaling or preprocessing.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
ITERATION_LIMIT_REACHED	The maximum allowable number of iterations was reached.
TIME_LIMIT_REACHED	The maximum time limit was reached.
FAILED	The solver failed to converge, possibly due to numerical issues.
NONCONVEX	The quadratic matrix is nonconvex (minimization).
NONCONCAVE	The quadratic matrix is nonconcave (maximization).

OBJECTIVE
indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY
indicates the (relative) infeasibility of the primal constraints at the solution. See the section "Interior Point Algorithm: Overview" on page 698 for details.

DUAL_INFEASIBILITY
indicates the (relative) infeasibility of the dual constraints at the solution. See the section "Interior Point Algorithm: Overview" on page 698 for details.

BOUND_INFEASIBILITY
indicates the (relative) violation by the solution of the lower or upper bounds (or both). See the section "Interior Point Algorithm: Overview" on page 698 for details.

DUALITY_GAP
indicates the (relative) duality gap. See the section "Interior Point Algorithm: Overview" on page 698 for details.

COMPLEMENTARITY
indicates the (absolute) complementarity at the solution. See the section "Interior Point Algorithm: Overview" on page 698 for details.

ITERATIONS
indicates the number of iterations required to solve the problem.

PRESOLVE_TIME
indicates the time taken for preprocessing (in seconds).

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time that is reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

Examples: OPTQP Procedure

This section contains examples that illustrate the use of the OPTQP procedure. Example 14.1 illustrates how to model a linear least squares problem and solve it by using PROC OPTQP. Example 14.2 and Example 14.3 explain in detail how to model the portfolio optimization and selection problems.

Example 14.1: Linear Least Squares Problem

The linear least squares problem arises in the context of determining a solution to an overdetermined set of linear equations. In practice, these equations could arise in data fitting and estimation problems. An overdetermined system of linear equations can be defined as

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, and $m > n$. Since this system usually does not have a solution, you need to be satisfied with some sort of approximate solution. The most widely used approximation is the least squares solution, which minimizes $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$.

This problem is called a least squares problem for the following reason. Let $\mathbf{A}$, $\mathbf{x}$, and $\mathbf{b}$ be defined as previously. Let $k_i(x)$ be the kth component of the vector $\mathbf{A}\mathbf{x} - \mathbf{b}$:

$$k_i(x) = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - b_i, \ i = 1, 2, \ldots, m$$

By definition of the Euclidean norm, the objective function can be expressed as follows:

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 = \sum_{i=1}^{m} k_i(x)^2$$

Therefore, the function you minimize is the sum of squares of m terms $k_i(x)$; hence the term least squares. The following example is an illustration of the *linear* least squares problem; that is, each of the terms k_i is a linear function of x.function $\sum_{ij} a_{ij}x_j$ plus a constant, $-b_i$.

Consider the following least squares problem defined by

$$\mathbf{A} = \begin{bmatrix} 4 & 0 \\ -1 & 1 \\ 3 & 2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

This translates to the following set of linear equations:

$$4x_1 = 1, \quad -x_1 + x_2 = 0, \quad 3x_1 + 2x_2 = 1$$

The corresponding least squares problem is

$$\text{minimize} \quad (4x_1 - 1)^2 + (-x_1 + x_2)^2 + (3x_1 + 2x_2 - 1)^2$$

The preceding objective function can be expanded to

$$\text{minimize} \quad 26x_1^2 + 5x_2^2 + 10x_1x_2 - 14x_1 - 4x_2 + 2$$

In addition, you impose the following constraint so that the equation $3x_1 + 2x_2 = 1$ is satisfied within a tolerance of 0.1:

$$0.9 \leq 3x_1 + 2x_2 \leq 1.1$$

You can create the QPS-format input data set by using the following SAS statements:

```
data lsdata;
   input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
   datalines;
NAME      .       LEASTSQ      .             .          .
ROWS      .       .            .             .          .
N         OBJ     .            .             .          .
G         EQ3     .            .             .          .
COLUMNS   .       .            .             .          .
.         X1      OBJ         -14            EQ3        3
.         X2      OBJ         -4             EQ3        2
RHS       .       .            .             .          .
.         RHS     OBJ         -2             EQ3        0.9
RANGES    .       .            .             .          .
.         RNG     EQ3          0.2           .          .
BOUNDS    .       .            .             .          .
FR        BND1    X1           .             .          .
FR        BND1    X2           .             .          .
QUADOBJ   .       .            .             .          .
.         X1      X1           52            .          .
.         X1      X2           10            .          .
.         X2      X2           10            .          .
ENDATA    .       .            .             .          .
;
```

The decision variables x_1 and x_2 are free, so they have bound type FR in the BOUNDS section of the QPS-format data set.

You can use the following SAS statements to solve the least squares problem:

```
proc optqp data=lsdata
   printlevel = 0
   primalout = lspout;
run;
```

The optimal solution is displayed in Output 14.1.1.

Output 14.1.1 Solution to the Least Squares Problem

Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Linear Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status
1	OBJ	RHS	X1	F	-14	-1.7977E308	1.7977E308	0.23810	O
2	OBJ	RHS	X2	F	-4	-1.7977E308	1.7977E308	0.16190	O

The iteration log is shown in Output 14.1.2.

Output 14.1.2 Iteration Log

```
NOTE: The problem LEASTSQ has 2 variables (2 free, 0 fixed).
NOTE: The problem has 1 constraints (0 LE, 0 EQ, 0 GE, 1 range).
NOTE: The problem has 2 constraint coefficients.
NOTE: The objective function has 2 Hessian diagonal elements and 1 Hessian
      elements above the diagonal.
NOTE: The QP presolver value AUTOMATIC is applied.
NOTE: The QP presolver removed 0 variables and 0 constraints.
NOTE: The QP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 2 variables, 1 constraints, and 2 constraint
      coefficients.
NOTE: The QP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Interior Point algorithm is using up to 4 threads.
                                        Primal       Bound        Dual
      Iter  Complement Duality Gap      Infeas       Infeas       Infeas   Time
         0  1.9181E-02  5.8936E-03  1.9637E-08  0.0000E+00  3.5390E-04      0
         1  9.0486E-04  2.8311E-04  8.6896E-10  1.1565E-17  1.3055E-05      0
         2  1.5370E-05  4.9441E-06  6.4151E-11  2.3130E-17  1.3055E-07      0
         3  1.5357E-07  4.9397E-08  1.7428E-12  5.7824E-18  1.3056E-09      0
NOTE: Optimal.
NOTE: Objective = 0.0095238095.
NOTE: The Interior Point solve time is 0.02 seconds.
NOTE: The data set WORK.LSPOUT has 2 observations and 9 variables.
```

Example 14.2: Portfolio Optimization

Consider a portfolio optimization example. The two competing goals of investment are (1) long-term growth of capital and (2) low risk. A good portfolio grows steadily without wild fluctuations in value. The Markowitz model is an optimization model for balancing the return and risk of a portfolio. The decision variables are the amounts invested in each asset. The objective is to minimize the variance of the portfolio's total return, subject to the constraints that (1) the expected growth of the portfolio reaches at least some target level and (2) you do not invest more capital than you have.

Let $x_1, \ldots, x_n$ be the amount invested in each asset, $\mathcal{B}$ be the amount of capital you have, $\mathbf{R}$ be the random vector of asset returns over some period, and $\mathbf{r}$ be the expected value of $\mathbf{R}$. Let G be the minimum growth you hope to obtain, and $\mathcal{C}$ be the covariance matrix of $\mathbf{R}$. The objective function is $\mathrm{Var}\left(\sum_{i=1}^{n} x_i R_i\right)$, which can be equivalently denoted as $\mathbf{x}^{\mathrm{T}}\mathcal{C}\mathbf{x}$.

Assume, for example, $n = 4$. Let $\mathcal{B} = 10{,}000$, $G = 1000$, $\mathbf{r} = [0.05, -0.2, 0.15, 0.30]$, and

$$\mathcal{C} = \begin{bmatrix} 0.08 & -0.05 & -0.05 & -0.05 \\ -0.05 & 0.16 & -0.02 & -0.02 \\ -0.05 & -0.02 & 0.35 & 0.06 \\ -0.05 & -0.02 & 0.06 & 0.35 \end{bmatrix}$$

The QP formulation can be written as follows:

$$\begin{aligned}\min \quad & 0.08x_1^2 - 0.1x_1x_2 - 0.1x_1x_3 - 0.1x_1x_4 + 0.16x_2^2 \\ & -0.04x_2x_3 - 0.04x_2x_4 + 0.35x_3^2 + 0.12x_3x_4 + 0.35x_4^2\end{aligned}$$

$$\begin{array}{lrcl}\text{subject to} & & & \\ \text{(budget)} & x_1 + x_2 + x_3 + x_4 & \leq & 10000 \\ \text{(growth)} & 0.05x_1 - 0.2x_2 + 0.15x_3 + 0.30x_4 & \geq & 1000 \\ & x_1,\ x_2,\ x_3,\ x_4 & \geq & 0\end{array}$$

The corresponding QPS-format input data set is as follows:

```
data portdata;
   input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
datalines;
NAME     .          PORT       .          .          .
ROWS     .          .          .          .          .
N        OBJ.FUNC   .          .          .          .
L        BUDGET     .          .          .          .
G        GROWTH     .          .          .          .
COLUMNS  .          .          .          .          .
.        X1         BUDGET     1.0        GROWTH     0.05
.        X2         BUDGET     1.0        GROWTH     -.20
.        X3         BUDGET     1.0        GROWTH     0.15
.        X4         BUDGET     1.0        GROWTH     0.30
RHS      .          .          .          .          .
.        RHS        BUDGET     10000      .          .
.        RHS        GROWTH     1000       .          .
RANGES   .          .          .          .          .
BOUNDS   .          .          .          .          .
QUADOBJ  .           .         .           .         .
.        X1         X1         0.16       .          .
.        X1         X2         -.10       .          .
.        X1         X3         -.10       .          .
.        X1         X4         -.10       .          .
.        X2         X2         0.32       .          .
.        X2         X3         -.04       .          .
.        X2         X4         -.04       .          .
.        X3         X3         0.70       .          .
.        X3         X4         0.12       .          .
.        X4         X4         0.70       .          .
ENDATA   .          .          .          .          .
;
```

Use the following SAS statements to solve the problem:

```
proc optqp data=portdata
  primalout = portpout
  printlevel = 0
  dualout   = portdout;
run;
```

The optimal solution is shown in Output 14.2.1.

Output 14.2.1 Portfolio Optimization

The OPTQP Procedure
Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Linear Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status
1	OBJ.FUNC	RHS	X1	N	0	0	1.7977E308	3452.86	O
2	OBJ.FUNC	RHS	X2	N	0	0	1.7977E308	0.00	O
3	OBJ.FUNC	RHS	X3	N	0	0	1.7977E308	1068.81	O
4	OBJ.FUNC	RHS	X4	N	0	0	1.7977E308	2223.45	O

Thus, the minimum variance portfolio that earns an expected return of at least 10% is $x_1 = 3452.86$, $x_2 = 0$, $x_3 = 1068.81$, $x_4 = 2223.45$. Asset 2 gets nothing, because its expected return is -20% and its covariance with the other assets is not sufficiently negative for it to bring any diversification benefits. What if you drop the nonnegativity assumption? You need to update the BOUNDS section in the existing QPS-format data set to indicate that the decision variables are free.

```
...
RANGES    .          .          .          .          .
BOUNDS    .          .          .          .          .
FR        BND1       X1         .          .          .
FR        BND1       X2         .          .          .
FR        BND1       X3         .          .          .
FR        BND1       X4         .          .          .
QUADOBJ   .          .          .          .          .
...
```

Financially, that means you are allowed to short-sell—that is, sell low-mean-return assets and use the proceeds to invest in high-mean-return assets. In other words, you put a negative portfolio weight in low-mean assets and "more than 100%" in high-mean assets. You can see in the optimal solution displayed in Output 14.2.2 that the decision variable x_2, denoting Asset 2, is equal to -1563.61, which means short sale of that asset.

Output 14.2.2 Portfolio Optimization with Short-Sale Option

The OPTQP Procedure
Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Linear Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status
1	OBJ.FUNC	RHS	X1	F	0	-1.7977E308	1.7977E308	1684.35	O
2	OBJ.FUNC	RHS	X2	F	0	-1.7977E308	1.7977E308	-1563.61	O
3	OBJ.FUNC	RHS	X3	F	0	-1.7977E308	1.7977E308	682.51	O
4	OBJ.FUNC	RHS	X4	F	0	-1.7977E308	1.7977E308	1668.95	O

Example 14.3: Portfolio Selection with Transactions

Consider a portfolio selection problem with a slight modification. You are now required to take into account the current position and transaction costs associated with buying and selling assets. The objective is to find the minimum variance portfolio. In order to understand the scenario better, consider the following data.

You are given three assets. The current holding of the three assets is denoted by the vector **c** = [200, 300, 500], the amount of asset bought and sold is denoted by b_i and s_i, respectively, and the net investment in each asset is denoted by x_i and is defined by the following relation:

$$x_i - b_i + s_i = c_i, \quad i = 1, 2, 3$$

Suppose you pay a transaction fee of 0.01 every time you buy or sell. Let the covariance matrix $\mathcal{C}$ be defined as

$$\mathcal{C} = \begin{bmatrix} 0.027489 & -0.00874 & -0.00015 \\ -0.00874 & 0.109449 & -0.00012 \\ -0.00015 & -0.00012 & 0.000766 \end{bmatrix}$$

Assume that you hope to obtain at least 12% growth. Let **r** = [1.109048, 1.169048, 1.074286] be the vector of expected return on the three assets, and let $\mathcal{B}$=1000 be the available funds. Mathematically, this problem can be written in the following manner:

$$\begin{aligned} \min \quad & 0.027489x_1^2 - 0.01748x_1x_2 - 0.0003x_1x_3 + 0.109449x_2^2 \\ & -0.00024x_2x_3 + 0.000766x_3^2 \end{aligned}$$

subject to

$$\begin{array}{lrcll} \text{(return)} & \sum_{i=1}^{3} r_i x_i & \geq & 1.12\mathcal{B} & \\ \text{(budget)} & \sum_{i=1}^{3} x_i + \sum_{i=1}^{3} 0.01(b_i + s_i) & = & \mathcal{B} & \\ \text{(balance)} & x_i - b_i + s_i & = & c_i, & i = 1, 2, 3 \\ & x_i,\ b_i,\ s_i & \geq & 0, & i = 1, 2, 3 \end{array}$$

The QPS-format input data set is as follows:

```
data potrdata;
   input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
datalines;
NAME     .         POTRAN    .           .          .
ROWS     .         .         .           .          .
N        OBJ.FUNC  .         .           .          .
G        RETURN    .         .           .          .
E        BUDGET    .         .           .          .
E        BALANC1   .         .           .          .
E        BALANC2   .         .           .          .
E        BALANC3   .         .           .          .
COLUMNS  .         .         .           .          .
.        X1        RETURN    1.109048    BUDGET     1.0
.        X1        BALANC1   1.0         .          .
.        X2        RETURN    1.169048    BUDGET     1.0
.        X2        BALANC2   1.0         .          .
.        X3        RETURN    1.074286    BUDGET     1.0
.        X3        BALANC3   1.0         .          .
.        B1        BUDGET    .01         BALANC1   -1.0
.        B2        BUDGET    .01         BALANC2   -1.0
.        B3        BUDGET    .01         BALANC3   -1.0
.        S1        BUDGET    .01         BALANC1    1.0
.        S2        BUDGET    .01         BALANC2    1.0
.        S3        BUDGET    .01         BALANC3    1.0
RHS      .         .         .           .          .
.        RHS       RETURN    1120        .          .
.        RHS       BUDGET    1000        .          .
.        RHS       BALANC1   200         .          .
.        RHS       BALANC2   300         .          .
.        RHS       BALANC3   500         .          .
RANGES   .         .         .           .          .
BOUNDS   .         .         .           .          .
QUADOBJ  .         .         .           .          .
.        X1        X1        0.054978    .          .
.        X1        X2        -.01748     .          .
.        X1        X3        -.0003      .          .
.        X2        X2        0.218898    .          .
.        X2        X3        -.00024     .          .
.        X3        X3        0.001532    .          .
ENDATA   .         .         .           .          .
;
```

Use the following SAS statements to solve the problem:

```
proc optqp data=potrdata
  primalout = potrpout
  printlevel = 0
  dualout   = potrdout;
run;
```

The optimal solution is displayed in Output 14.3.1.

Output 14.3.1 Portfolio Selection with Transactions

The OPTQP Procedure
Primal Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Linear Objective Coefficient	Lower Bound	Upper Bound	Variable Value	Variable Status
1	OBJ.FUNC	RHS	X1	N	0	0	1.7977E308	397.584	O
2	OBJ.FUNC	RHS	X2	N	0	0	1.7977E308	406.115	O
3	OBJ.FUNC	RHS	X3	N	0	0	1.7977E308	190.165	O
4	OBJ.FUNC	RHS	B1	N	0	0	1.7977E308	197.584	O
5	OBJ.FUNC	RHS	B2	N	0	0	1.7977E308	106.115	O
6	OBJ.FUNC	RHS	B3	N	0	0	1.7977E308	0.000	O
7	OBJ.FUNC	RHS	S1	N	0	0	1.7977E308	0.000	O
8	OBJ.FUNC	RHS	S2	N	0	0	1.7977E308	0.000	O
9	OBJ.FUNC	RHS	S3	N	0	0	1.7977E308	309.835	O

References

Freund, R. W. (1991). "On Polynomial Preconditioning and Asymptotic Convergence Factors for Indefinite Hermitian Matrices." *Linear Algebra and Its Applications* 154–156:259–288.

Freund, R. W., and Jarre, F. (1997). "A QMR-Based Interior Point Algorithm for Solving Linear Programs." *Mathematical Programming* 76:183–210.

Freund, R. W., and Nachtigal, N. M. (1996). "QMRPACK: A Package of QMR Algorithms." *ACM Transactions on Mathematical Software* 22:46–77.

Vanderbei, R. J. (1999). "LOQO: An Interior Point Code for Quadratic Programming." *Optimization Methods and Software* 11:451–484.

Wright, S. J. (1997). *Primal-Dual Interior-Point Methods*. Philadelphia: SIAM.

Chapter 15
The Decomposition Algorithm

Contents

Overview: Decomposition Algorithm **716**
Getting Started: Decomposition Algorithm **718**
Solving a MILP with DECOMP and PROC OPTMODEL 718
Solving a MILP with DECOMP and PROC OPTMILP 720
Syntax: Decomposition Algorithm **721**
Decomposition Algorithm Options in the PROC OPTLP Statement or the SOLVE WITH LP Statement in PROC OPTMODEL 722
Decomposition Algorithm Options in the PROC OPTMILP Statement or the SOLVE WITH MILP Statement in PROC OPTMODEL 723
DECOMP Statement 726
DECOMP_MASTER Statement 732
DECOMP_MASTER_IP Statement 734
DECOMP_SUBPROB Statement 736
Details: Decomposition Algorithm **741**
Data Input 741
Decomposition Algorithm 742
Parallel Processing 743
Special Case: Identical Blocks and Ryan-Foster Branching 743
Log for the Decomposition Algorithm 747
Examples: Decomposition Algorithm **749**
Example 15.1: Multicommodity Flow Problem and METHOD=NETWORK 749
Example 15.2: Generalized Assignment Problem 755
Example 15.3: Block-Diagonal Structure and METHOD=CONCOMP in Single-Machine Mode 762
Example 15.4: Block-Diagonal Structure and METHOD=CONCOMP in Distributed Mode 766
Example 15.5: Block-Angular Structure and METHOD=AUTO 769
Example 15.6: Bin Packing Problem 773
Example 15.7: Resource Allocation Problem 778
Example 15.8: Vehicle Routing Problem 791
Example 15.9: ATM Cash Management in Single-Machine Mode 797
Example 15.10: ATM Cash Management in Distributed Mode 808
Example 15.11: Kidney Donor Exchange and METHOD=SET 811
References **819**

Overview: Decomposition Algorithm

The SAS/OR decomposition algorithm (DECOMP) provides an alternative method of solving linear programs (LPs) and mixed integer linear programs (MILPs) by exploiting the ability to efficiently solve a relaxation of the original problem. The algorithm is available as an option in the OPTMODEL, OPTLP, and OPTMILP procedures and is based on the methodology described in Galati (2009).

A standard linear or mixed integer linear program has the formulation

$$
\begin{array}{lllllll}
\text{minimize} & \mathbf{c}^\top\mathbf{x} & + & \mathbf{f}^\top\mathbf{y} & & & \\
\text{subject to} & \mathbf{D}\mathbf{x} & + & \mathbf{B}\mathbf{y} & \{\geq,=,\leq\} & \mathbf{d} & \text{(master)} \\
 & \mathbf{A}\mathbf{x} & & & \{\geq,=,\leq\} & \mathbf{b} & \text{(subproblem)} \\
 & \underline{x}_i \leq x_i \leq \overline{x}_i, & & x_i \in \mathbb{Z} & i \in \mathcal{S}_x & & \\
 & \underline{y}_i \leq y_i \leq \overline{y}_i, & & y_i \in \mathbb{Z} & i \in \mathcal{S}_y & &
\end{array}
$$

where

$\mathbf{x} \in \mathbb{R}^n$	is the vector of structural variables
$\mathbf{y} \in \mathbb{R}^s$	is the vector of master-only structural variables
$\mathbf{c} \in \mathbb{R}^n$	is the vector of objective function coefficients that are associated with variables $\mathbf{x}$
$\mathbf{f} \in \mathbb{R}^s$	is the vector of objective function coefficients that are associated with variables $\mathbf{y}$
$\mathbf{D} \in \mathbb{R}^{t\times n}$	is the matrix of master constraint coefficients that are associated with variables $\mathbf{x}$
$\mathbf{B} \in \mathbb{R}^{t\times s}$	is the matrix of master constraint coefficients that are associated with variables $\mathbf{y}$
$\mathbf{A} \in \mathbb{R}^{m\times n}$	is the matrix of subproblem constraint coefficients
$\mathbf{d} \in \mathbb{R}^t$	is the vector of master constraints' right-hand sides
$\mathbf{b} \in \mathbb{R}^m$	is the vector of subproblem constraints' right-hand sides
$\underline{\mathbf{x}} \in \mathbb{R}^n$	is the vector of lower bounds on variables $\mathbf{x}$
$\overline{\mathbf{x}} \in \mathbb{R}^n$	is the vector of upper bounds on variables $\mathbf{x}$
$\underline{\mathbf{y}} \in \mathbb{R}^s$	is the vector of lower bounds on variables $\mathbf{y}$
$\overline{\mathbf{y}} \in \mathbb{R}^s$	is the vector of upper bounds on variables $\mathbf{y}$
$\mathcal{S}_x$	is a subset of the set $\{1,\ldots,n\}$ of indices on variables $\mathbf{x}$
$\mathcal{S}_y$	is a subset of the set $\{1,\ldots,s\}$ of indices on variables $\mathbf{y}$

You can form a relaxation of the preceding mathematical program by removing the master constraints, which are defined by the matrices $\mathbf{D}$ and $\mathbf{B}$. The resulting constraint system, defined by the matrix $\mathbf{A}$, forms the subproblem, which can often be solved much more efficiently than the entire original problem. This is one of the key motivators for using the decomposition algorithm.

The decomposition algorithm works by finding convex combinations of extreme points of the subproblem polyhedron that satisfy the constraints defined in the master. For MILP subproblems, the strength of the relaxation is another important motivator for using this method. If the subproblem polyhedron defines feasible solutions that are close to the original feasible space, the chance of success for the algorithm increases.

The region that defines the subproblem space is often separable. That is, the formulation of the preceding mathematical program can be written in *block-angular* form as

$$
\begin{array}{lccccccccccc}
\text{minimize} & \mathbf{c}^1\mathbf{x}^1 & + & \mathbf{c}^2\mathbf{x}^2 & + & \cdots & + & \mathbf{c}^\kappa\mathbf{x}^\kappa & + & \mathbf{f}^\top y & & \\
\text{subject to} & \mathbf{D}^1\mathbf{x}^1 & + & \mathbf{D}^2\mathbf{x}^2 & + & \cdots & + & \mathbf{D}^\kappa\mathbf{x}^\kappa & + & \mathbf{B}\mathbf{y} & \{\geq,=,\leq\} & \mathbf{d} \\
 & \mathbf{A}^1\mathbf{x}^1 & & & & & & & & & \{\geq,=,\leq\} & \mathbf{b}^1 \\
 & & & \mathbf{A}^2\mathbf{x}^2 & & & & & & & \{\geq,=,\leq\} & \mathbf{b}^2 \\
 & & & & & \ddots & & & & & \{\geq,=,\leq\} & \vdots \\
 & & & & & & & \mathbf{A}^\kappa\mathbf{x}^\kappa & & & \{\geq,=,\leq\} & \mathbf{b}^\kappa \\
 & \underline{x}_i \leq x_i \leq \overline{x}_i, \quad x_i \in \mathbb{Z} \quad i \in \mathcal{S}_x & & & & & & & & & & \\
 & \underline{y}_i \leq y_i \leq \overline{y}_i, \quad y_i \in \mathbb{Z} \quad i \in \mathcal{S}_y & & & & & & & & & &
\end{array}
$$

where $K = \{1, \ldots, \kappa\}$ defines a partition of the constraints (and variables) into independent subproblems (blocks) such that $\mathbf{A} = [\mathbf{A}^1 \ldots \mathbf{A}^\kappa]$, $\mathbf{D} = [\mathbf{D}^1 \ldots \mathbf{D}^\kappa]$, $\mathbf{c} = [\mathbf{c}^1 \ldots \mathbf{c}^\kappa]$, $\mathbf{b} = [\mathbf{b}^1 \ldots \mathbf{b}^\kappa]$, $\underline{\mathbf{x}} = [\underline{\mathbf{x}}^1 \ldots \underline{\mathbf{x}}^\kappa]$, $\overline{\mathbf{x}} = [\overline{\mathbf{x}}^1 \ldots \overline{\mathbf{x}}^\kappa]$, and $\mathbf{x} = [\mathbf{x}^1 \ldots \mathbf{x}^\kappa]$. This type of structure is relatively common in modeling mathematical programs. For example, consider a model that defines a workplace that has separate departmental restrictions (defined as the subproblem constraints), which are coupled together by a company-wide budget across departments (defined as the master constraint). By relaxing the budget (master) constraint, the decomposition algorithm can take advantage of the fact that the decoupled subproblems are separable, and it can process them in parallel. A special case of block-angular form, called *block-diagonal* form, occurs when the set of master constraints is empty. In this special case, the subproblem matrices define the entire original problem.

An important indicator of a problem that is well suited for decomposition is the amount by which the subproblems cover the original problem with respect to both variables and constraints in the original presolved model. This value, which is expressed as a percentage of the original model, is known as the *coverage*. For LPs, the decomposition algorithm usually performs better than standard approaches only if the subproblems cover a significant amount of the original problem. For MILPs, the correlation between performance and coverage is more difficult to determine, because the strength of the subproblem with respect to integrality is not always proportional to the size of the system. Regardless, it is unlikely that the decomposition algorithm will outperform more standard methods (such as branch-and-cut) in problems that have small coverage.

The primary input and output for the decomposition algorithm are identical to those that are needed and produced by the OPTLP, OPTMILP, and OPTMODEL procedures. For more information, see the following sections:

- "Data Input and Output" on page 582 in Chapter 12, "The OPTLP Procedure"
- "Data Input and Output" on page 641 in Chapter 13, "The OPTMILP Procedure"
- "Details: LP Solver" on page 266 in Chapter 7, "The Linear Programming Solver"
- "Details: MILP Solver" on page 336 in Chapter 8, "The Mixed Integer Linear Programming Solver"

The only additional input that can be provided for the decomposition algorithm is an explicit definition of the partition of the subproblem constraints. The following section gives a simple example of providing this input for both PROC OPTMILP and PROC OPTMODEL.

Getting Started: Decomposition Algorithm

This example illustrates how you can use the decomposition algorithm to solve a simple mixed integer linear program. Suppose you want to solve the following problem:

$$
\begin{array}{rlllllllllllll}
\max & x_{11} & + & 2x_{21} & + & x_{31} & + & & + & x_{22} & + & x_{32} & & \\
\text{subject to} & x_{11} & & & & & & x_{12} & & & & & \geq & 1 & \text{(m)} \\
& 5x_{11} & + & 7x_{21} & + & 4x_{31} & & & & & & & \leq & 11 & \text{(s1)} \\
& & & & & & & x_{12} & + & 2x_{22} & + & x_{32} & \leq & 2 & \text{(s2)} \\
& x_{ij} \in \{0,1\} \;\; i \in \{1,\ldots,3\}, \;\; j \in \{1,\ldots,2\} & & & & & & & & & & & & &
\end{array}
$$

It is obvious from the structure of the problem that if constraint m is removed, then the remaining constraints s1 and s2 decompose into two independent subproblems. The next two sections describe how to solve this MILP by using the decomposition algorithm in the OPTMODEL procedure and OPTMILP procedure, respectively.

Solving a MILP with DECOMP and PROC OPTMODEL

The following statements use the OPTMODEL procedure and the decomposition algorithm to solve the MILP:

```
proc optmodel;
   var x{i in 1..3, j in 1..2} binary;

   max f =    x[1,1] + 2*x[2,1] +   x[3,1]
                     +   x[2,2] +   x[3,2];

   con m :    x[1,1] +   x[1,2]            >=  1;
   con s1:  5*x[1,1] + 7*x[2,1] + 4*x[3,1] <= 11;
   con s2:    x[1,2] + 2*x[2,2] +   x[3,2] <=  2;

   s1.block = 0;
   s2.block = 1;

   solve with milp / presolver=none decomp=(logfreq=1);
   print x;
quit;
```

Here, the PRESOLVER=NONE option is used, because otherwise the presolver solves this small instance without invoking any solver. The solution summary and optimal solution are displayed in Figure 15.1.

Figure 15.1 Solution Summary and Optimal Solution

The OPTMODEL Procedure

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	f
Solution Status	Optimal
Objective Value	4
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	0
Best Bound	4
Nodes	1
Iterations	1
Presolve Time	0.00
Solution Time	0.03

	x	
	1	**2**
1	0	1
2	1	0
3	1	1

The iteration log, which displays the problem statistics, the progress of the solution, and the optimal objective value, is shown in Figure 15.2.

Figure 15.2 Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 6 binary and 0 integer variables.
NOTE: The problem has 3 linear constraints (2 LE, 0 EQ, 1 GE, 0 range).
NOTE: The problem has 8 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The number of block threads has been reduced to 2 threads.
NOTE: The problem has a decomposable structure with 2 blocks. The largest block covers 33.33%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 6 (100%) variables and 2 (66.67%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best       Master          Best       LP        IP  CPU Real
                   Bound    Objective       Integer      Gap       Gap Time Time
         Node  Active   Sols         Best          Best      Gap    CPU   Real
                                   Integer         Bound            Time   Time
            0       0      1       4.0000       4.0000    0.00%      0      0
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 0.03 seconds.
NOTE: Optimal.
NOTE: Objective = 4.
```

Solving a MILP with DECOMP and PROC OPTMILP

Alternatively, to solve the MILP with the OPTMILP procedure, create a corresponding SAS data set that uses the mathematical programming system (MPS) format as follows:

```
data mpsdata;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME     .         mpsdata   .         .         .
ROWS     .         .         .         .         .
MAX      f         .         .         .         .
G        m         .         .         .         .
L        s1        .         .         .         .
L        s2        .         .         .         .
COLUMNS  .         .         .         .         .
```

```
.       .MRK0000 'MARKER' .        'INTORG' .
.       x[1,1]   f        1        m        1
.       x[1,1]   s1       5        .        .
.       x[2,1]   f        2        s1       7
.       x[3,1]   f        1        s1       4
.       x[1,2]   m        1        s2       1
.       x[2,2]   f        1        s2       2
.       x[3,2]   f        1        s2       1
.       .MRK0001 'MARKER' .        'INTEND' .
RHS     .        .        .        .        .
.       .RHS.    m        1        .        .
.       .RHS.    s1       11       .        .
.       .RHS.    s2       2        .        .
BOUNDS  .        .        .        .        .
UP      .BOUNDS. x[1,1]   1        .        .
UP      .BOUNDS. x[2,1]   1        .        .
UP      .BOUNDS. x[3,1]   1        .        .
UP      .BOUNDS. x[1,2]   1        .        .
UP      .BOUNDS. x[2,2]   1        .        .
UP      .BOUNDS. x[3,2]   1        .        .
ENDATA  .        .        .        .        .
;
```

Next, use the following SAS data set to define the subproblem blocks:

```
data blocks;
   input _row_ $ _block_;
   datalines;
s1 0
s2 1
;
```

Now, you can use the following OPTMILP statements to solve this MILP:

```
proc optmilp
   data       = mpsdata
   presolver  = none;
   decomp
      logfreq = 1
      blocks  = blocks;
run;
```

Syntax: Decomposition Algorithm

You can specify the decomposition algorithm either by using options in a SOLVE statement in the OPTMODEL procedure or by using statements in the OPTLP and OPTMILP procedures. Except for the fact that you use SOLVE statement options in PROC OPTMODEL or you use statements in PROC OPTLP and PROC OPTMILP, the syntax is identical.

The following decomposition algorithm options are available in the SOLVE statement in the OPTMODEL procedure:

SOLVE WITH LP / < *options* >
< **DECOMP**< *=(decomp-options)* > >
< **DECOMP_MASTER=(**< *decomp-master-options* >**)** >
< **DECOMP_SUBPROB=(**< *decomp-subprob-options* >**)** > **;**

SOLVE WITH MILP / < *options* >
< **DECOMP**< *=(decomp-options)* > >
< **DECOMP_MASTER=(**< *decomp-master-options* >**)** >
< **DECOMP_MASTER_IP=(**< *decomp-master-ip-options* >**)** >
< **DECOMP_SUBPROB=(**< *decomp-subprob-options* >**)** > **;**

The following statements are available in the OPTLP procedure:

PROC OPTLP < *options* > **;**
DECOMP < *decomp-options* > **;**
DECOMP_MASTER < *decomp-master-options* > **;**
DECOMP_SUBPROB < *decomp-subprob-options* > **;**

The following statements are available in the OPTMILP procedure:

PROC OPTMILP < *options* > **;**
DECOMP < *decomp-options* > **;**
DECOMP_MASTER < *decomp-master-options* > **;**
DECOMP_MASTER_IP < *decomp-master-ip-options* > **;**
DECOMP_SUBPROB < *decomp-subprob-options* > **;**

Decomposition Algorithm Options in the PROC OPTLP Statement or the SOLVE WITH LP Statement in PROC OPTMODEL

To solve a linear program, you can specify the decomposition algorithm in a SOLVE WITH LP statement in the OPTMODEL procedure or in a PROC OPTLP statement in the OPTLP procedure. To control the overall decomposition algorithm, you can specify one or more of the LP solver options that are shown in Table 15.1. (As indicated, you can specify some options only in the PROC OPTLP statement.)

The options in Table 15.1 control the overall process flow for solving a linear program, and they are equivalent to the options that are used in PROC OPTLP and PROC OPTMODEL with standard methods. These options are called main solver options in this chapter. They are described in detail in the section "Syntax: LP Solver" on page 259 in Chapter 7, "The Linear Programming Solver," and the section "Syntax: OPTLP Procedure" on page 574 in Chapter 12, "The OPTLP Procedure." The DUALIZE= option has a different default when you use the decomposition algorithm, as shown in Table 15.1.

Table 15.1 LP Options in the PROC OPTLP Statement or SOLVE WITH LP Statement

Description	*option*	Different Default
Data Set Options (OPTLP procedure only)		
Specifies the input data set	DATA=	
Specifies the dual solution output data set	DUALOUT=	
Specifies whether the model is a maximization or minimization problem	OBJSENSE=	
Specifies the primal solution output data set	PRIMALOUT=	

Table 15.1 (continued)

Description	*option*	Different Default
Saves output data sets only if optimal	SAVE_ONLY_IF_OPTIMAL	
Presolve Options		
Controls the dualization of the problem	DUALIZE=	OFF
Specifies the type of presolve	PRESOLVER=	
Control Options		
Specifies the feasibility tolerance	FEASTOL=	
Specifies how frequently to print the solution progress	LOGFREQ=	
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=	
Specifies the maximum number of iterations	MAXITER=	
Specifies the time limit for the optimization process	MAXTIME=	
Specifies the optimality tolerance	OPTTOL=	
Enables or disables printing summary (OPTLP procedure only)	PRINTLEVEL=	
Specifies whether time units are CPU time or real time	TIMETYPE=	
Algorithm Options		
Enables or disables scaling of the problem	SCALE=	

Decomposition Algorithm Options in the PROC OPTMILP Statement or the SOLVE WITH MILP Statement in PROC OPTMODEL

To solve a mixed integer linear program, you can specify the decomposition algorithm in a SOLVE WITH MILP statement in the OPTMODEL procedure or in a PROC OPTMILP statement in the OPTMILP procedure. To control the overall decomposition algorithm, you can specify one or more of the MILP solver options shown in Table 15.2. (As indicated, you can specify some options only in the PROC OPTMILP statement.)

The options in Table 15.2 control the overall process flow for solving a mixed integer linear program, and they are equivalent to the options that are used in the OPTMILP and OPTMODEL procedures with standard methods. These options are called main solver options in this chapter. They are described in detail in the section "Syntax: MILP Solver" on page 325 and the section "Syntax: OPTMILP Procedure" on page 629.

The HYBRID= option in the DECOMP statement indicates the processing mode for the root node of the branch-and-bound search tree. When HYBRID=ON, the root node is first processed using standard MILP techniques, as described in the section "Details: MILP Solver" on page 336. The default setting for the decomposition algorithm is HYBRID=OFF. In this case, the root processing is done solely by the decomposition algorithm, and several of the direct MILP options are ignored. These options are indicated in Table 15.2 in the column Ignored HYBRID=OFF.

Table 15.2 MILP Options in the PROC OPTMILP Statement or SOLVE WITH MILP Statement

Description	*option*	Ignored HYBRID=OFF
Data Set Options (OPTMILP procedure only)		
Specifies the input data set	DATA=	
Specifies the constraint activities output data set	DUALOUT=	
Specifies whether the model is a maximization or minimization problem	OBJSENSE=	
Specifies the primal solution input data set (warm start)	PRIMALIN=	
Specifies the primal solution output data set	PRIMALOUT=	
Presolve Option		
Specifies the type of presolve	PRESOLVER=	
Control Options		
Specifies the stopping criterion based on an absolute objective gap	ABSOBJGAP=	
Emphasizes feasibility or optimality	EMPHASIS=	X
Specifies the maximum violation of variables and constraints	FEASTOL=	
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=	
Specifies how frequently to print the node log	LOGFREQ=	
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=	
Specifies the maximum number of nodes to be processed	MAXNODES=	
Specifies the maximum number of solutions to be found	MAXSOLS=	
Specifies the time limit for the optimization process	MAXTIME=	
Specifies the tolerance used in determining the optimality of nodes in the branch-and-bound tree	OPTTOL=	
Uses the input primal solution (warm start) (OPTMODEL procedure only)	PRIMALIN	
Enables or disables printing summary (OPTMILP procedure only)	PRINTLEVEL=	
Specifies the probing level	PROBE=	
Specifies the stopping criterion based on a relative objective gap	RELOBJGAP=	
Specifies the scale of the problem matrix	SCALE=	
Specifies the initial seed for the random number generator	SEED=	X
Specifies the stopping criterion based on target objective value	TARGET=	X
Specifies whether time units are CPU time or real time	TIMETYPE=	
Heuristics Option		
Specifies the primal heuristics level	HEURISTICS=	
Search Options		
Specifies the number of pricing iterations performed on each variable in the strong branching strategy	STRONGITER=	
Specifies the number of candidates for strong branching	STRONGLEN=	
Specifies the level of symmetry detection	SYMMETRY=	
Specifies the rule for selecting the branching variable	VARSEL=	

Table 15.2 (continued)

Description	*option*	Ignored HYBRID=OFF
Cut Options		
Specifies the cut level for all cuts	ALLCUTS=	X
Specifies the clique cut level	CUTCLIQUE=	X
Specifies the flow cover cut level	CUTFLOWCOVER=	X
Specifies the flow path cut level	CUTFLOWPATH=	X
Specifies the Gomory cut level	CUTGOMORY=	X
Specifies the generalized upper bound (GUB) cover cut level	CUTGUB=	X
Specifies the implied bounds cut level	CUTIMPLIED=	X
Specifies the knapsack cover cut level	CUTKNAPSACK=	X
Specifies the lift-and-project cut level	CUTLAP=	X
Specifies the mixed lifted 0-1 cut level	CUTMILIFTED=	X
Specifies the mixed integer rounding (MIR) cut level	CUTMIR=	X
Specifies the multicommodity network flow cut level	CUTMULTICOMMODITY=	X
Specifies the row multiplier factor for cuts	CUTSFACTOR=	X
Specifies the overall cut aggressiveness	CUTSTRATEGY=	X
Specifies the zero-half cut level	CUTZEROHALF=	X

The following search options, listed in Table 15.2, have a different interpretation from what is described in the MILP solver sections.

LOGFREQ=*number*

PRINTFREQ=*number*

specifies how often information is printed in the node log. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is 10. If you set *number* to 0, then the node log is disabled. If *number* is positive, then an entry is made in the node log at the first node, at the last node, and at intervals dictated by the value of *number*. An entry is also made in the node log each time the solver finds a better integer solution or improved bound.

STRONGITER=*number* | **AUTOMATIC**

specifies the number of pricing iterations that are performed for each variable in the candidate list when you use the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, the MILP solver uses the default value, which is calculated automatically.

The following search option, listed in Table 15.2, has a different set of options from what is described in the MILP solver sections.

VARSEL=*number* | *string*

specifies the rule for selecting the branching variable. The values of *string* and the corresponding values of *number* are listed in Table 15.3.

Table 15.3 Values for VARSEL= Option

number	*string*	Description
–1	AUTOMATIC	Uses automatic branching variable selection.

Table 15.3 (continued)

number	*string*	**Description**
0	MAXINFEAS	Selects the variable in the original compact formulation with maximum infeasibility.
2	PSEUDO	Selects the variable in the original compact formulation that maximizes the weighted up and down pseudocosts.
3	STRONG	Selects the variable in the original compact formulation that maximizes the estimated improvement in the objective value based on strong branching.
4	RYANFOSTER	When appropriate, uses a specialized branching rule known as *Ryan-Foster branching*.

The default value is AUTOMATIC. For more information about variable selection, see the sections "Variable Selection" on page 645 and "Special Case: Identical Blocks and Ryan-Foster Branching" on page 743.

DECOMP Statement

DECOMP < *decomp-options* > **;**

The DECOMP statement controls the overall decomposition algorithm.

Table 15.4 summarizes the *decomp-options* available in the DECOMP statement. These options control the overall decomposition algorithm process flow during the solution of an LP or a MILP. (As indicated, you can specify the data set options only in the OPTLP or OPTMILP procedure, and you can specify some control options only for a MILP.)

Table 15.4 Options in the DECOMP Statement

Description	*decomp-option*
Data Set Options (OPTLP and OPTMILP procedures only)	
Specifies the blocks input data set	BLOCKS=
Control Options	
Specifies the stopping criterion based on an absolute objective gap	ABSOBJGAP=
Specifies the frequency of removing ineffective columns from the master LP	COMPRESSFREQ=
Specifies whether or not to first process the root node by using standard MILP techniques	HYBRID=
Specifies whether to initialize the columns by solving each block with the original cost vector	INITVARS=
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=
Specifies the maximum number of blocks to allow	MAXBLOCKS=
Specifies the maximum number of new columns to allow into the master each pass	MAXCOLSPASS=
Specifies the maximum amount of time spent in the decomposition algorithm	MAXTIME=

Table 15.4 (continued)

Description	*decomp-option*
Specifies the decomposition algorithm method	METHOD=
Specifies the number of blocks to search for by using METHOD=AUTO	NBLOCKS=
Specifies the number of block threads to use in the decomposition algorithm	NTHREADS=
Specifies the stopping criterion based on relative objective gap	RELOBJGAP=
Control Options (MILP only)	
Specifies how frequently to print the continuous iteration log	LOGFREQ=
Specifies whether the master problem is solved as a MILP with the current set of columns at the beginning of phase II	MASTER_IP_BEG=
Specifies whether the master problem is solved as a MILP with the current set of columns at the end of phase II	MASTER_IP_END=
Specifies the frequency of solving the master problem as a MILP with the current set of columns	MASTER_IP_FREQ=
Specifies the maximum number of outer iterations for the decomposition algorithm	MAXITER=

The following list describes the *decomp-options* in detail.

ABSOBJGAP=*number*

specifies a stopping criterion for the continuous bound of the decomposition. When the absolute difference between the master objective and the best dual bound falls below the value of *number*, the decomposition algorithm stops adding columns. The value of *number* can be any nonnegative number. The default value is the value of the OPTTOL= main solver option.

BLOCKS=*SAS-data-set*

specifies (for OPTLP and OPTMILP procedures only) the input data set that contains block definitions to use in the decomposition algorithm if METHOD=USER. See the section "The BLOCKS= Data Set in PROC OPTMILP and PROC OPTLP" on page 741 for more information. To specify blocks in PROC OPTMODEL, use the `.block` constraint suffix instead (see the section "The `.block` Constraint Suffix in PROC OPTMODEL" on page 741).

COMPRESSFREQ=*number*

removes ineffective columns from the master LP after every *number* of iterations. The frequency, *number*, is an integer between 0 and the largest four-byte signed integer, which is $2^{31} - 1$. The default value is 0.

HYBRID=*number* | *string*

specifies whether to first process the root node by using standard MILP techniques, as described in the section "Details: MILP Solver" on page 336.

Table 15.5 describes the valid values of the HYBRID= option.

Table 15.5 Values for HYBRID= Option

number	*string*	Description
0	OFF	Disables root processing by standard MILP techniques.
1	ON	Enables root processing by standard MILP techniques.

The default is OFF.

INITVARS=*number* | *string*
specifies whether to initialize the columns by using the original cost vector to solve each block.

Table 15.6 describes the valid values of the INITVARS= option.

Table 15.6 Values for INITVARS= Option

number	*string*	**Description**
0	OFF	Disables initializing the columns by using the original cost vector to solve each block.
1	ON	Enables initializing the columns by using the original cost vector to solve each block.

This option must be set to ON when used with METHOD=CONCOMP. The default is ON.

LOGFREQ=*number*
specifies (for MILP problems only) how often to print information in the continuous iteration log. The value of *number* can be any nonnegative number up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of *number* is 10. If *number* is set to 0, then the iteration log is disabled. If *number* is positive, then an entry is made in the log at the first iteration, at the last iteration, and at intervals that are dictated by the value of *number*. An entry is also made each time a better integer solution or improved bound is found.

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log by the decomposition algorithm. Table 15.7 and Table 15.8 provide the valid values for this option and a description of what is displayed in the log when an LP and a MILP, respectively, is solved.

Table 15.7 Values for LOGLEVEL= Option for an LP

number	*string*	**Description**
–1	AUTOMATIC	Prints the continuous iteration log at the interval dictated by the LOGFREQ= main solver option.
0	NONE	Turns off printing of all of the decomposition algorithm messages to the SAS log.
1	BASIC	Prints the continuous iteration log at the interval dictated by the LOGFREQ= main solver option.
2	MODERATE	Prints the continuous iteration log and summary information for each iteration at the interval dictated by the LOGFREQ= main solver option.
3	AGGRESSIVE	Prints the continuous iteration log and detailed information for each iteration at the interval dictated by the LOGFREQ= main solver option.

Table 15.8 Values for LOGLEVEL= Option for a MILP

number	*string*	**Description**
–1	AUTOMATIC	Prints the continuous iteration log for the root node at the interval dictated by the LOGFREQ= option in the DECOMP statement. Prints the branch-and-bound node log at the interval dictated by the LOGFREQ= main solver option.
0	NONE	Turns off printing of all of the decomposition algorithm messages to the SAS log.
1	BASIC	Prints the continuous iteration log for each branch-and-bound node at the interval dictated by the LOGFREQ= option in the DECOMP statement.
2	MODERATE	Prints the continuous iteration log and summary information for each iteration of each branch-and-bound node at the interval dictated by the LOGFREQ= option in the DECOMP statement.
3	AGGRESSIVE	Prints the continuous iteration log and detailed information for each iteration of each branch-and-bound node at the interval dictated by the LOGFREQ= option in the DECOMP statement.

The default is AUTOMATIC for both LPs and MILPs.

MASTER_IP_BEG=*number* | *string*

specifies (for MILP problems only) whether the master problem is solved as a MILP with the current set of columns at the beginning of phase II. Table 15.9 describes the valid values of the MASTER_IP_BEG= option.

Table 15.9 Values for MASTER_IP_BEG= Option

number	*string*	**Description**
0	OFF	Disables solving the master as a MILP at the beginning of phase II.
1	ON	Enables solving the master as a MILP at the beginning of phase II.

The default is ON in the root node and automatically determines whether to call the heuristic in the branch-and-bound tree.

MASTER_IP_END=*number* | *string*

specifies (for MILP problems only) whether the master problem is solved as a MILP with the current set of columns at the end of phase II. Table 15.10 describes the valid values of the MASTER_IP_END= option.

Table 15.10 Values for MASTER_IP_END= Option

number	*string*	**Description**
0	OFF	Disables solving the master as a MILP at the end of phase II.
1	ON	Enables solving the master as a MILP at the end of phase II.

The default is ON in the root node and automatically determines whether to call the heuristic in the branch-and-bound tree.

MASTER_IP_FREQ=*number*
solves the master problem (for MILP problems only) as a MILP with the current set of columns after every *number* iterations. The frequency, *number*, is an integer between 0 and the largest four-byte signed integer, which is $2^{31} - 1$. The default is 10 in the root node and 0 elsewhere.

MAXBLOCKS=*number*
specifies the maximum number of blocks to allow. If the defined number of blocks exceeds *number*, the algorithm creates superblocks using a very simple round-robin scheme. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

MAXCOLSPASS=*number*
specifies the maximum number of new columns to allow into the master at each pass. This option is disabled on the initial pass if INITVARS=1. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

MAXITER=*number*
specifies (for MILP problems only) the maximum number of outer iterations for the decomposition algorithm. The value *number* can be any integer between 1 and the largest four-byte signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed.

MAXTIME=*number*
specifies an upper limit of *number* seconds of time for the decomposition algorithm. The value of the TIMETYPE= main solver option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

METHOD=*string*
specifies the decomposition algorithm method as shown in Table 15.11.

Table 15.11 Values for METHOD= Option

string	**Description**
AUTO	The algorithm attempts to find a block-angular structure in the constraint matrix by using matrix-stretching techniques similar to what is described in Grcar (1990) and Aykanat, Pinar, and Çatalyürek (2004). The NBLOCKS= option specifies the number of blocks into which the algorithm attempts to decompose the constraint matrix. If the algorithm fails to find a decomposition, the MILP solver is called directly.

string	**Description**
CONCOMP	The algorithm attempts to find a block-diagonal (not block-angular) structure in the constraint matrix. Unless your problem separates into completely independent problems with no linking constraints, this method finds only one block and hence is equivalent to calling the MILP solver directly.
NETWORK	The algorithm attempts to find an embedded network similar to what is described in the section "The Network Simplex Algorithm" on page 266. The weakly connected components of this network are used as the blocks.
SET	The algorithm attempts to find a set partitioning or set covering structure in the constraint matrix and defines this as the master (linking) constraints. The weakly connected components of the remaining constraints are used as the blocks.
USER	The user defines which rows belong to which blocks (subproblems). In PROC OPTMODEL, use the `.block` constraint suffix. In PROC OPTLP and PROC OPTMILP, use the BLOCKS= data set instead.

The default is USER if blocks are defined and AUTO otherwise.

NBLOCKS=*number*

specifies the initial number of blocks to search for when you specify METHOD=AUTO. If the algorithm is unable to find a block-angular structure that contains this number of blocks, it repeatedly attempts to find an appropriate structure that contains half the previously attempted number of blocks. If the algorithm fails to find a decomposition that contains at least two blocks, then the standard MILP solver is called directly. The value of *number* can be any positive number less than or equal to the number of rows in the presolved model; the default value is the number of block threads that are used for processing. In single-machine mode, this is equivalent to the value of the NTHREADS= option in the DECOMP statement. In distributed mode, this is equivalent to the number of compute nodes that you specify in the NODES= option in the PERFORMANCE statement times the number of threads that you specify for each compute node. For more information about parallel execution, see the section "Parallel Processing" on page 743.

NTHREADS=*number*

specifies the number of block threads to use in the decomposition algorithm. The value of the NTHREADS= option in the PERFORMANCE statement, which is described in the section "PERFORMANCE Statement" on page 21, serves as the overall capacity for the number of active threads that can run at one time. By default, the number of block threads is $t = \min(p, d, b)$, where p is the value of the NTHREADS= option in the PERFORMANCE statement, d is the value of the NTHREADS= option in the DECOMP statement, and b is the number of blocks that the decomposition algorithm sets of finds.

RELOBJGAP=*number*

specifies the relative objective gap as a stopping criterion. The relative objective gap is based on the master objective (MasterObjective) and the best dual bound (BestBound); it is equal to

$$| \text{MasterObjective} - \text{BestBound} | / (1E{-}10 + | \text{BestBound} |)$$

When this value becomes smaller than the specified gap size *number*, the decomposition algorithm stops adding columns. The value of *number* can be any nonnegative number. For LP, the default value is 0; for MILP, the default value is 1e-4.

DECOMP_MASTER Statement

DECOMP_MASTER < *decomp-master-options* > **;**

MASTER < *decomp-master-options* > **;**

The DECOMP_MASTER statement controls the master problem.

Table 15.12 summarizes the options available in the DECOMP_MASTER statement. These options control the master LP solver in the decomposition algorithm during the solution of an LP or a MILP. (As indicated, you can specify the PRINTLEVEL= option only in the OPTLP procedure.) For descriptions of these options, see the section "LP Solver Options" on page 260 in Chapter 7, "The Linear Programming Solver," and the section "PROC OPTLP Statement" on page 576 in Chapter 12, "The OPTLP Procedure." Some options have different defaults when you use the decomposition algorithm, as indicated in Table 15.12.

Table 15.12 Options in the DECOMP_MASTER Statement

Description	*decomp-master-option*	Different Default
Algorithm Option		
Specifies the master algorithm	ALGORITHM=	PS†
Presolve Option		
Controls the dualization of the problem	DUALIZE=	OFF
Specifies, for the first master solve only, the type of presolve	INITPRESOLVER=	
Specifies the type of presolve	PRESOLVER=	NONE (ALGORITHM=PS) †
Control Options		
Specifies the feasibility tolerance	FEASTOL=	1E–7
Specifies how frequently to print the solution progress	LOGFREQ=	
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=	
Specifies the maximum number of iterations	MAXITER=	
Specifies the time limit for the optimization process	MAXTIME=	
Specifies the number of threads to use in the master solver	NTHREADS=	
Specifies the optimality tolerance	OPTTOL=	1E–7
Enables or disables printing summary (OPTLP procedure only)	PRINTLEVEL=	
Specifies whether time units are CPU time or real time	TIMETYPE=	

Table 15.12 (continued)

Description	*decomp-master-option*	Different Default
Specifies the type of initial basis	BASIS=	WARMSTART (ALGORITHM=PS)†
Specifies the type of pricing strategy	PRICETYPE=	
Specifies the queue size for determining the entering variable	QUEUESIZE=	
Enables or disables scaling of the problem	SCALE=	
Specifies the initial seed for the random number generator	SEED=	
Interior Point Algorithm Options		
Enables or disables interior crossover	CROSSOVER=	
Specifies the stopping criterion based on a duality gap	STOP_DG=	
Specifies the stopping criterion based on dual infeasibility	STOP_DI=	
Specifies the stopping criterion based on primal infeasibility	STOP_PI=	

† The different defaults (ALGORITHM=PS, PRESOLVER=NONE, and BASIS=WARMSTART) are motivated by the fact that primal feasibility of the master problem is preserved when columns are added, so a warm start from the previous optimal basis tends to be more efficient than solving the master from scratch in each iteration.

The following options, listed in Table 15.12, are specific to the DECOMP_MASTER statement and are not described in the LP solver sections.

INITPRESOLVER=*number* | *string*

INITPRESOL=*number* | *string*

specifies, for the first master solve only, presolve conditions as shown in Table 15.13.

Table 15.13 Values for INITPRESOLVER= Option

number	*string*	Description
–1	AUTOMATIC	Applies the default level of presolve processing.
0	NONE	Disables presolver.
1	BASIC	Performs minimal presolve processing.
2	MODERATE	Applies a higher level of presolve processing.
3	AGGRESSIVE	Applies the highest level of presolve processing.

The default is AUTOMATIC.

NTHREADS=*number*

specifies the number of threads to use in the master solver (if the selected solver method supports multithreading). The value of the NTHREADS= option in the PERFORMANCE statement, which is described in the section "PERFORMANCE Statement" on page 21, serves as the overall capacity for the number of active threads that can run at one time. By default, the number of master threads is $t = \min(p, m)$, where p is the value of the NTHREADS= option in the PERFORMANCE statement and m is the value of the NTHREADS= option in the DECOMP_MASTER statement.

DECOMP_MASTER_IP Statement

DECOMP_MASTER_IP < *decomp-master-ip-options* > **;**

MASTER_IP < *decomp-master-ip-options* > **;**

For mixed integer linear programming problems, the DECOMP_MASTER_IP statement controls the (restricted) master problem, which is solved as a MILP with the current set of columns in an effort to obtain an integer-feasible solution.

Table 15.14 summarizes the options available in the DECOMP_MASTER_IP statement. These options control the MILP solver that is used to solve the integer version of the master problem. For descriptions of these options, see the section "MILP Solver Options" on page 327 in Chapter 8, "The Mixed Integer Linear Programming Solver," and the section "PROC OPTMILP Statement" on page 630 in Chapter 13, "The OPTMILP Procedure." Some options have different defaults when you use the decomposition algorithm, as shown in Table 15.14.

Table 15.14 Options in the DECOMP_MASTER_IP Statement

Description	*decomp-master-ip-option*	Different Default
Presolve Option		
Specifies the type of presolve	PRESOLVER=	
Control Options		
Specifies the stopping criterion based on an absolute objective gap	ABSOBJGAP=	
Specifies the cutoff value for node removal	CUTOFF=	
Emphasizes feasibility or optimality	EMPHASIS=	
Specifies the maximum violation on variables and constraints	FEASTOL=	1E–7
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=	
Specifies how frequently to print the node log	LOGFREQ=	
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=	
Specifies the maximum number of nodes to be processed	MAXNODES=†	
Specifies the maximum number of solutions to be found	MAXSOLS=	
Specifies the time limit for the optimization process	MAXTIME=	
Specifies the number of threads to use in the master integer solver	NTHREADS=	
Specifies the tolerance used when deciding on the optimality of nodes in the branch-and-bound tree	OPTTOL=	1E–7
Specifies whether to use the previous best primal solution as a warm start	PRIMALIN=	
Specifies the probing level	PROBE=	
Specifies the stopping criterion based on a relative objective gap	RELOBJGAP=	0.01
Specifies the scale of the problem matrix	SCALE=	
Specifies the stopping criterion based on the target objective value	TARGET=	

Table 15.14 (continued)

Description	*decomp-master-ip-option*	Different Default
Specifies whether time units are CPU time or real time	TIMETYPE=	
Heuristics Option		
Specifies the primal heuristics level	HEURISTICS=	
Search Options		
Specifies the level of conflict search	CONFLICTSEARCH=	
Specifies the node selection strategy	NODESEL=	
Specifies the restarting strategy	RESTARTS=	
Specifies the initial seed for the random number generator	SEED=	
Specifies the number of simplex iterations performed on each variable in strong branching strategy	STRONGITER=	
Specifies the number of candidates for strong branching	STRONGLEN=	
Specifies the level of symmetry detection	SYMMETRY=	
Specifies the rule for selecting branching variable	VARSEL=	
Cut Options		
Specifies the cut level for all cuts	ALLCUTS=	
Specifies the clique cut level	CUTCLIQUE=	
Specifies the flow cover cut level	CUTFLOWCOVER=	
Specifies the flow path cut level	CUTFLOWPATH=	
Specifies the Gomory cut level	CUTGOMORY=	
Specifies the generalized upper bound (GUB) cover cut level	CUTGUB=	
Specifies the implied bounds cut level	CUTIMPLIED=	
Specifies the knapsack cover cut level	CUTKNAPSACK=	
Specifies the lift-and-project cut level	CUTLAP=	
Specifies the mixed lifted 0-1 cut level	CUTMILIFTED=	
Specifies the mixed integer rounding (MIR) cut level	CUTMIR=	
Specifies the multicommodity network flow cut level	CUTMULTICOMMODITY=	
Specifies the row multiplier factor for cuts	CUTSFACTOR=	
Specifies the overall cut aggressiveness	CUTSTRATEGY=	
Specifies the zero-half cut level	CUTZEROHALF=	

† MAXNODES=100000 in the root node, and MAXNODES=10000 in nodes that are not the root.

The following options are listed in Table 15.14 but are not described in the MILP solver sections. These options are specific to the DECOMP_MASTER_IP statement.

NTHREADS=*number*

specifies the number of threads to use in the master integer solver (if the selected solver method supports multithreading). The value of the NTHREADS= option in the PERFORMANCE statement, which is described in the section "PERFORMANCE Statement" on page 21, serves as the overall capacity for the number of active threads that can run at one time. By default, the number of master integer threads is $t = \min(p, m)$, where p is the value of the NTHREADS= option in the PERFORMANCE statement and m is the value of the NTHREADS= option in the DECOMP_MASTER_IP statement.

PRIMALIN=*number* | *string*

PIN=*number* | *string*

specifies whether the MILP solver is to use the previous best solution's variables values as a starting solution (warm start). If the MILP solver finds that the input solution is feasible, then the input solution provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution is not feasible, the MILP solver tries to repair it. When it is difficult to find a good integer-feasible solution for a problem, warm start can reduce solution time significantly. Table 15.15 describes the valid values of the PRIMALIN= option.

Table 15.15 Values for PRIMALIN= Option

number	*string*	**Description**
0	OFF	Ignores the previous solution.
1	ON	Starts from the previous solution.

The default is ON.

DECOMP_SUBPROB Statement

DECOMP_SUBPROB < *decomp-subprob-options* > **;**

SUBPROB < *decomp-subprob-options* > **;**

The DECOMP_SUBPROB statement controls the subproblem.

Table 15.16 summarizes the options available for the decomposition algorithm in the DECOMP_SUBPROB statement when the subproblem algorithm chosen is an LP algorithm. (As indicated, you can specify the PRINTLEVEL= option only in the OPTLP procedure.) For descriptions of these options, see the section "LP Solver Options" on page 260 in Chapter 7, "The Linear Programming Solver," and the section "PROC OPTLP Statement" on page 576 in Chapter 12, "The OPTLP Procedure." Some options have different defaults when you use the decomposition algorithm, as shown in Table 15.16.

Table 15.16 Options in the DECOMP_SUBPROB Statement Used with an LP Algorithm

Description	*decomp-subprob-option*	**Different Default**
Algorithm Option		
Specifies the subproblem algorithm	ALGORITHM=	PS (METHOD=USER) NETWORK_PURE (METHOD=NETWORK)†
Presolve Option		
Controls the dualization of the problem	DUALIZE=	OFF
Specifies, for the first subproblem solve only, the type of presolve	INITPRESOLVER=	
Specifies the type of presolve	PRESOLVER=	NONE (ALGORITHM=PS)†
Control Options		
Specifies the feasibility tolerance	FEASTOL=	1E–7

Table 15.16 (continued)

Description	*decomp-subprob-option*	Different Default
Specifies how frequently to print the solution progress	LOGFREQ=	
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=	
Specifies the maximum number of iterations	MAXITER=	
Specifies the time limit for the optimization process	MAXTIME=	
Specifies the number of threads to use in the subproblem solver	NTHREADS=	
Specifies the optimality tolerance	OPTTOL=	1E–7
Enables or disables printing summary (OPTLP procedure only)	PRINTLEVEL=	
Specifies the initial seed for the random number generator	SEED=	
Specifies whether time units are CPU time or real time	TIMETYPE=	
Simplex Algorithm Options		
Specifies the type of initial basis	BASIS=	WARMSTART (ALGORITHM=PS)†
Specifies the type of pricing strategy	PRICETYPE=	
Specifies the queue size for determining entering variable	QUEUESIZE=	
Enables or disables scaling of the problem	SCALE=	
Interior Point Algorithm Options		
Enables or disables interior crossover	CROSSOVER=	
Specifies the stopping criterion based on duality gap	STOP_DG=	
Specifies the stopping criterion based on dual infeasibility	STOP_DI=	
Specifies the stopping criterion based on primal infeasibility	STOP_PI=	

† When METHOD=USER is specified in the DECOMP statement, ALGORITHM=PS, PRESOLVER=NONE, and BASIS=WARMSTART by default. These defaults are motivated by the fact that primal feasibility of the subproblem is preserved when the objective is changed, so a warm start from the previous optimal basis tends to be more efficient than solving the subproblem from scratch in each iteration. When METHOD=NETWORK, ALGORITHM=NETWORK_PURE by default because each subproblem is a pure network, causing the specialized pure network solver to usually be the most efficient choice.

Table 15.17 summarizes the options available in the DECOMP_SUBPROB statement when the subproblem algorithm chosen is a MILP algorithm. When the subproblem consists of multiple blocks (a block-diagonal structure), these settings apply to all subproblems. For descriptions of these options, see the section "MILP Solver Options" on page 327 in Chapter 8, "The Mixed Integer Linear Programming Solver," and the section "PROC OPTMILP Statement" on page 630 in Chapter 13, "The OPTMILP Procedure."

Table 15.17 Options in the DECOMP_SUBPROB Statement Used with a MILP Algorithm

Description	*decomp-subprob-option*	Different Default
Algorithm Option		
Specifies the subproblem algorithm	ALGORITHM=	
Presolve Option		
Specifies, for the first subproblem solve only, the type of presolve	INITPRESOLVER=	
Specifies the type of presolve	PRESOLVER=	
Control Options		
Specifies the stopping criterion based on absolute objective gap	ABSOBJGAP=	
Emphasizes feasibility or optimality	EMPHASIS=	
Specifies the maximum violation on variables and constraints	FEASTOL=	1E–7
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=	
Specifies how frequently to print the node log	LOGFREQ=	
Specifies the level of detail of solution progress to print in the log	LOGLEVEL=	
Specifies the maximum number of nodes to be processed	MAXNODES=	
Specifies the maximum number of solutions to be found	MAXSOLS=	
Specifies the time limit for the optimization process	MAXTIME=	
Specifies the number of threads to use in the subproblem solver	NTHREADS=	
Specifies the tolerance used when deciding on the optimality of nodes in the branch-and-bound tree	OPTTOL=	1E–7
Specifies whether to use the previous best primal solution as a warm start	PRIMALIN=	
Specifies the probing level	PROBE=	
Specifies the stopping criterion based on relative objective gap	RELOBJGAP=	
Specifies the scale of the problem matrix	SCALE=	
Specifies the stopping criterion based on target objective value	TARGET=	
Specifies whether time units are CPU time or real time	TIMETYPE=	
Heuristics Option		
Specifies the primal heuristics level	HEURISTICS=	
Search Options		
Specifies the level of conflict search	CONFLICTSEARCH=	
Specifies the node selection strategy	NODESEL=	
Specifies the restarting strategy	RESTARTS=	

Table 15.17 (continued)

Description	*decomp-subprob-option*	Different Default
Specifies the initial seed for the random number generator	SEED=	
Specifies the number of simplex iterations performed on each variable in strong branching strategy	STRONGITER=	
Specifies the number of candidates for strong branching	STRONGLEN=	
Specifies the level of symmetry detection	SYMMETRY=	
Specifies the rule for selecting branching variable	VARSEL=	
Cut Options		
Specifies the cut level for all cuts	ALLCUTS=	
Specifies the clique cut level	CUTCLIQUE=	
Specifies the flow cover cut level	CUTFLOWCOVER=	
Specifies the flow path cut level	CUTFLOWPATH=	
Specifies the Gomory cut level	CUTGOMORY=	
Specifies the generalized upper bound (GUB) cover cut level	CUTGUB=	
Specifies the implied bounds cut level	CUTIMPLIED=	
Specifies the knapsack cover cut level	CUTKNAPSACK=	
Specifies the lift-and-project cut level	CUTLAP=	
Specifies the mixed lifted 0-1 cut level	CUTMILIFTED=	
Specifies the mixed integer rounding (MIR) cut level	CUTMIR=	
Specifies the multicommodity network flow cut level	CUTMULTICOMMODITY=	
Specifies the row multiplier factor for cuts	CUTSFACTOR=	
Specifies the overall cut aggressiveness	CUTSTRATEGY=	
Specifies the zero-half cut level	CUTZEROHALF=	

The following options, listed in Table 15.16 and Table 15.17, are specific to the DECOMP_SUBPROB statement and are not described in the LP or MILP solver sections.

ALGORITHM=*string*

SOLVER=*string*

SOL=*string*

specifies one of the algorithms shown in Table 15.18 (the valid abbreviated value for each *string* is shown in parentheses).

Table 15.18 Values for ALGORITHM= Option

string	Description
PRIMAL (PS)	Uses the primal simplex algorithm.
DUAL (DS)	Uses the dual simplex algorithm.
NETWORK (NS)	Uses the network simplex algorithm.

string	**Description**
NETWORK_PURE (NSPURE)	Uses the network simplex algorithm for pure networks.
INTERIORPOINT (IP)	Uses the interior point algorithm.
MILP	Uses the mixed integer linear solver.

The default is NETWORK_PURE if METHOD=NETWORK, MILP for mixed integer linear programming subproblems, or PS for linear programming subproblems.

INITPRESOLVER=*number* | *string*

INITPRESOL=*number* | *string*

specifies, for the first subproblem solve only, presolve conditions as listed in Table 15.19.

Table 15.19 Values for INITPRESOLVER= Option

number	*string*	**Description**
–1	AUTOMATIC	Applies the default level of presolve processing
0	NONE	Disables presolver
1	BASIC	Performs minimal presolve processing
2	MODERATE	Applies a higher level of presolve processing
3	AGGRESSIVE	Applies the highest level of presolve processing

The default is AUTOMATIC.

NTHREADS=*number*

specifies the number of threads to use in the subproblem solver (if the selected solver method supports multithreading). The value of the NTHREADS= option in the PERFORMANCE statement, which is described in the section "PERFORMANCE Statement" on page 21, serves as the overall capacity for the number of active threads that can run at one time. By default, the number of subproblem threads is $t = \max(1, \min(s, \lfloor p/n \rfloor))$, where s is the value of the NTHREADS= option in the DECOMP_SUBPROB statement, p is the value of the NTHREADS= option in the PERFORMANCE statement, $n = \max(1, \lfloor d/m \rfloor)$ is the number of blocks being processed simultaneously, d is the number of block threads, and m is the number of compute nodes (which can be more than one, when you run the decomposition algorithm in distributed mode).

PRIMALIN=*number* | *string*

PIN=*number* | *string*

specifies (for MILP problems only) whether the MILP solver is to use the values of the previous best solution's variables as a starting solution (warm start). If the MILP solver finds that the input solution is feasible, then the input solution provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution is not feasible, the MILP solver tries to repair it. When it is difficult to find a good integer-feasible solution for a problem, warm start can reduce solution time significantly. Table 15.20 describes the valid values of the PRIMALIN= option.

Table 15.20 Values for PRIMALIN= Option

number	*string*	**Description**
0	OFF	Ignores the previous solution.
1	ON	Starts from the previous solution.

The default is ON.

Details: Decomposition Algorithm

Data Input

This subsection describes the format for describing the partition of the constraint system that defines the subproblem blocks. In the OPTLP and OPTMILP procedures, partitioning is done by using a data set specified in the BLOCKS= data option in the DECOMP statement. In PROC OPTMODEL, partitioning is done by using the **.block** suffix on constraints.

The blocks must be disjoint with respect to variables. If two blocks contain a nonzero coefficient for the same variable, the decomposition algorithm produces an error that contains information about where the blocks overlap.

The BLOCKS= Data Set in PROC OPTMILP and PROC OPTLP

The BLOCKS= data set has two required variables:

ROW
specifies the constraint (row) names of the problem. The values should be a subset of the row names in the DATA= data set for the current problem.

BLOCK
specifies the numeric block identifier for each constraint in the problem. A missing observation or missing value indicates a master (linking) constraint that does not appear in any block. Listing the linking constraints is optional. The block identifiers must start from 0 and be consecutive.

See the section "Solving a MILP with DECOMP and PROC OPTMILP" on page 720 for an example of using this BLOCKS= data set with PROC OPTMILP.

The .block Constraint Suffix in PROC OPTMODEL

The **.block** constraint suffix specifies the numeric block identifier for each constraint in the problem. The block identifiers do not need to start from 0, nor do they need to be consecutive. Master (linking) constraints can be identified by using a missing value. Listing the linking constraints is optional.

See the section "Solving a MILP with DECOMP and PROC OPTMODEL" on page 718 for an example of using the **.block** constraint suffix with PROC OPTMODEL.

Decomposition Algorithm

The decomposition algorithm for LPs is based on the original Dantzig-Wolfe method (Dantzig and Wolfe 1960). Embedding this method in the context of a branch-and-bound algorithm for MILPs is described in Barnhart et al. (1998) and is often referred to as *branch-and-price*. The design of a framework that allows for building a generic branch-and-price solver that requires only the original (compact) formulation and the constraint partition was first proposed independently by Ralphs and Galati (2006) and Vanderbeck and Savelsbergh (2006). This method is also commonly referred to as *column generation*, although the algorithm implemented here is only one specific variant of this wider class of algorithms.

The algorithm setup starts by forming various components that are used iteratively during the solver process. These components include the master problem (controlled by options in the DECOMP_MASTER statement), one subproblem for each block (controlled by options in the DECOMP_SUBPROB statement) and, for MILPs, the integer version of the master problem (controlled by options in the DECOMP_MASTER_IP statement).

The master problem is a linear program that is defined over a potentially large number of variables that represent the weights of a convex combination. The points in the convex combination satisfy the constraints that are defined in the subproblem. The master constraints of the original problem are enforced in this reformulated space. In this sense, the decomposition algorithm takes the intersection of two polyhedra: one defined by original master constraints and one defined by the subproblem constraints. Since the set of variables needed to define the intersection of the polyhedra can be large, the algorithm works on a restricted subset and generates only those variables (columns) that have good potential with respect to feasibility and optimality. This generation is done by using the dual information that is obtained by solving the master problem to *price out* new variables. These new variables are generated by solving the subproblems over the appropriate cost vector (the reduced cost in the original space). This generation is similar to the revised simplex method, except that the variable space is exponentially large and therefore is generated implicitly by solving an optimization problem. This idea of generating variables as needed is the reason why this method is often referred to as *column generation*.

Similar to the two-phase simplex algorithm, the algorithm first introduces slack variables and solves a phase I problem to find a feasible solution. After the algorithm finds a feasible solution, it switches to a phase II problem to search for an optimal solution. The process of solving the master to generate pricing information and then solving one or more subproblems to generate candidate variables is repeated until there are no longer any improving variables and the method has converged.

For MILPs, this process is then used as a bounding method in a branch-and-bound algorithm, as described in the section "Branch-and-Bound Algorithm" on page 643. The strength of the subproblem polyhedron is one of the key reasons why decomposition can often solve problems that the standard branch-and-cut algorithm cannot solve in a reasonable amount of time. Since the points used in the convex combination are solutions (extreme points) of the subproblem (typically a MILP itself), then the bounds obtained can often be much stronger than the bounds obtained from standard branch-and-bound methods that are based on the LP relaxation. The subproblem polyhedron intersected with the continuous master polyhedron can provide a very good approximation of the true convex hull of the original integer program.

For more information about the algorithm process flow and the framework design, see Galati (2009).

Parallel Processing

At each iteration of the decomposition method, the subproblem is solved to minimize the reduced cost that is derived from the dual information that solving the master problem provides. As discussed in the section "Overview: Decomposition Algorithm" on page 716, the subproblem often has a block-diagonal structure that enables the solver to process each block independently.

You can run the decomposition algorithm in either a single-machine or a distributed computing environment. In single-machine mode, the computation is executed by multiple threads on a single computer. You can specify options for parallel execution in the PERFORMANCE statement, which is documented in the section "PERFORMANCE Statement" on page 21 of Chapter 4, "Shared Concepts and Topics." You can control the number of threads that are used by specifying the NTHREADS= option in the PERFORMANCE statement. In distributed mode, the computation is executed in a distributed computing environment. You can control the number of grid nodes (machines) that are used by specifying the NODES= option in the PERFORMANCE statement. The decomposition algorithm supports only the deterministic mode of the PARALLELMODE= option in the PERFORMANCE statement. The default mode of operation is single-machine mode, in which the number of concurrent threads is based on the number of CPUs (cores) on the machine (subject to any configuration limitations of the system).

The specified number of threads is used at each iteration to determine the number of blocks to be processed simultaneously. This same value also determines the number of threads to use for solving the master (continuous and integer) problem if the selected solver method supports multithreading. To avoid contention, the number of threads that are allocated to each subproblem solve is 1 (unless the number of blocks to process is less than the number of threads).

In addition, in each subcomponent statement you can use the NTHREADS= option to specify the number of threads to use for that solver.

NOTE: Distributed mode requires SAS High-Performance Optimization.

Special Case: Identical Blocks and Ryan-Foster Branching

In the special case of a set partitioning master problem and identical blocks, the underlying algorithm is automatically adjusted to reduce symmetry and improve overall performance. Identical blocks are subproblems (see the section "Overview: Decomposition Algorithm" on page 716) that have equivalent feasible regions (and optima) when they are projected. Algebraically, this means that

$$
\begin{array}{ccccccc}
\mathbf{A}^1 & = & \mathbf{A}^2 & = & \ldots & = & \mathbf{A}^\kappa \\
\mathbf{D}^1 & = & \mathbf{D}^2 & = & \ldots & = & \mathbf{D}^\kappa \\
\mathbf{c}^1 & = & \mathbf{c}^2 & = & \ldots & = & \mathbf{c}^\kappa \\
\mathbf{b}^1 & = & \mathbf{b}^2 & = & \ldots & = & \mathbf{b}^\kappa \\
\overline{\mathbf{x}}^1 & = & \overline{\mathbf{x}}^2 & = & \ldots & = & \overline{\mathbf{x}}^\kappa \\
\underline{\mathbf{x}}^1 & = & \underline{\mathbf{x}}^2 & = & \ldots & = & \underline{\mathbf{x}}^\kappa
\end{array}
$$

A *set partitioning* problem is a specific type of integer programming model in which each constraint represents choosing exactly one member of a set. These constraints are often referred to as *assignment constraints*. The linear relaxation of a set partitioning problem enables an algorithm to choose fractional parts of several members of some set such that they sum to 1. Algebraically, this means $\mathbf{Ax} = \mathbf{1}$, where all the coefficients in $\mathbf{A}$ are 0 or 1.

The performance of algorithms that use a branch-and-bound method can suffer when the formulation contains substructures that are symmetric. In this context, *symmetric* means that an assignment of solutions can be arbitrarily permuted for some component without affecting the optimality of that solution. For example, if

$$x_{11} = 1 \quad x_{12} = 0 \quad x_{21} = 0 \quad x_{22} = 1$$

and

$$x_{11} = 0 \quad x_{12} = 1 \quad x_{21} = 1 \quad x_{22} = 0$$

are both optimal, then these solutions, x_{ij}, are considered symmetric on index j. That is, you can interchange $j = 1$ and $j = 2$ without affecting the optimality of the solution. The presence of identical blocks in a mathematical program is an obvious case in which symmetry can hurt performance. In order to overcome this handicap, the decomposition algorithm aggregates the identical blocks into one block when it forms the Dantzig-Wolfe master problem. If the Dantzig-Wolfe master problem is a set partitioning model, the algorithm uses a specialized branching rule known as *Ryan-Foster branching*. If the original master model (after aggregation) is equivalent to the identity matrix, this guarantees that the Dantzig-Wolfe master problem is of the appropriate form. For more information about the aggregate formulation and Ryan-Foster branching, see Barnhart et al. (1998).

Suppose you want to solve the following problem:

$$\begin{array}{rrcrcrcrcrcrcll}
\max & x_{11} & + & 2x_{21} & + & x_{31} & + & x_{12} & + & 2x_{22} & + & x_{32} & & & \\
\text{subject to} & x_{11} & & & & & & x_{12} & & & & & = & 1 & \text{(m1)} \\
 & & & x_{21} & & & & & & x_{22} & & & = & 1 & \text{(m2)} \\
 & 5x_{11} & + & 7x_{21} & + & 4x_{31} & & & & & & & \leq & 11 & \text{(s1)} \\
 & & & & & & & 5x_{12} & + & 7x_{22} & + & 4x_{32} & \leq & 11 & \text{(s2)} \\
 & x_{ij} \in \{0,1\} \;\; i \in \{1,\ldots,3\}, \;\; j \in \{1,\ldots,2\} & & & & & & & & & & & & & &
\end{array}$$

If constraints m1 and m2 are removed, then the remaining constraints s1 and s2 decompose into two independent and identical subproblems. In addition, constraints m1 and m2 form a set partitioning master problem.

The following statements use the OPTMODEL procedure and the decomposition algorithm to solve the preceding problem:

```
proc optmodel;
   var x{i in 1..3, j in 1..2} binary;

   max f =    x[1,1] + 2*x[2,1] +   x[3,1] +
              x[1,2] + 2*x[2,2] +   x[3,2];

   con m1:    x[1,1] +   x[1,2]             =  1;
   con m2:    x[2,1] +   x[2,2]             =  1;
   con s1:  5*x[1,1] + 7*x[2,1] + 4*x[3,1] <= 11;
   con s2:  5*x[1,2] + 7*x[2,2] + 4*x[3,2] <= 11;

   s1.block = 0;
   s2.block = 1;

   solve with milp / presolver=none decomp=(logfreq=1);
   print x;
quit;
```

Here, the PRESOLVER=NONE option is used again, because otherwise the presolver solves this small instance without invoking any solver. The solution summary and optimal solution are displayed in Figure 15.3.

Figure 15.3 Solution Summary and Optimal Solution

The OPTMODEL Procedure

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	f
Solution Status	Optimal
Objective Value	5
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	0
Best Bound	5
Nodes	1
Iterations	2
Presolve Time	0.00
Solution Time	0.00

	x	
	1	**2**
1	1	0
2	0	1
3	1	1

The iteration log, which displays the problem statistics, the progress of the solution, and the optimal objective value, is shown in Figure 15.4.

Figure 15.4 Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 6 binary and 0 integer variables.
NOTE: The problem has 4 linear constraints (2 LE, 2 EQ, 0 GE, 0 range).
NOTE: The problem has 10 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: All blocks are identical and the master model is set partitioning.
NOTE: The Decomposition algorithm is using an aggregate formulation and Ryan-Foster branching.
NOTE: The number of block threads has been reduced to 1 threads.
NOTE: The problem has a decomposable structure with 2 blocks. The largest block covers 25% of
      the constraints in the problem.
NOTE: The decomposition subproblems cover 6 (100%) variables and 2 (50%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best       Master          Best       LP        IP  CPU Real
                   Bound    Objective       Integer      Gap       Gap Time Time
         .       6.0000       5.0000        5.0000   16.67%   16.67%    0    0
         1       6.0000       5.0000        5.0000   16.67%   16.67%    0    0
         2       5.0000       5.0000        5.0000    0.00%    0.00%    0    0
         Node  Active   Sols          Best          Best      Gap    CPU   Real
                                    Integer         Bound             Time   Time
            0       0      1       5.0000        5.0000    0.00%      0      0
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 0.00 seconds.
NOTE: Optimal.
NOTE: Objective = 5.
```

The decomposition solver recognizes that the original master model is of the appropriate form and that each block is identical. It formulates the aggregate master and uses Ryan-Foster branching to solve the model.

In the presence of identical blocks, under certain circumstances, the aggregate formulation can also be used with a set covering master formulation. A *set covering* problem is an integer programming model in which each constraint represents choosing at least one member of a set. Algebraically, this means $\mathbf{Ax} \geq \mathbf{1}$, where all the coefficients in $\mathbf{A}$ are 0 or 1. Aggregate formulation and Ryan-Foster branching can be used if there exists an optimal solution, $\mathbf{x}^*$, that is binding at equality ($\mathbf{Ax}^* = \mathbf{1}$). If you can guarantee such a condition, you can greatly improve performance by explicitly using VARSEL=RYANFOSTER as a MILP main solver option. The decomposition algorithm usually performs better when it uses a set covering formulation as opposed to a set partitioning formulation, because it is usually easier to find integer feasible solutions. If the models are equivalent, using the set covering formulation is recommended. For two examples, see Example 15.6, which shows the bin packing problem, and Example 15.8, which shows the vehicle routing problem.

Similarly, a *set packing* problem is an integer programming model in which each constraint represents choosing at most one member of a set. Algebraically, this means $\mathbf{Ax} \leq \mathbf{1}$, where all the coefficients in $\mathbf{A}$ are 0 or 1. Aggregate formulation and Ryan-Foster branching can be used if there exists an optimal solution, $\mathbf{x}^*$, that is binding at equality ($\mathbf{Ax}^* = \mathbf{1}$). In this case, using VARSEL=RYANFOSTER can improve performance. Alternatively, you can transform any set packing model into a set partitioning model by introducing a zero-cost slack variable for each packing constraint. See Example 15.11, which shows an application that optimizes a kidney donor exchange.

The decomposition algorithm automatically searches for identical blocks and the appropriate set partitioning master formulation. If it finds this structure, the algorithm automatically generates the aggregate formulation and uses Ryan-Foster branching. The aggregate model needs to process only one block at each subproblem iteration. Therefore, parallel processing (in which multiple blocks are processed simultaneously), as described in the section "Parallel Processing" on page 743, cannot improve performance. For this reason, when the decomposition algorithm runs in distributed mode, it does not create the aggregate formulation, nor does it use Ryan-Foster branching, even if the blocks are found to be identical.

Log for the Decomposition Algorithm

The following subsections describe what to expect in the SAS log when you run the decomposition algorithm.

Setup Information in the SAS Log

In the setup phase of the algorithm, information about the method you choose and the structure of the model is written to the SAS log. One of the most important pieces of information displayed in the log is the number of disjoint blocks and the coverage of those blocks with respect to both variables and constraints in the original presolved model. As explained in the section "Overview: Decomposition Algorithm" on page 716, the decomposition algorithm usually performs better than standard approaches only if the subproblems cover a significant amount of the original problem. However, this is not always a straightforward indicator for MILPs, because the strength of the subproblem with respect to integrality is not always proportional to the size of the system.

After the structural information is written to the log, the algorithm begins and the iteration log is displayed.

Iteration Log for LPs

When the decomposition algorithm solves LPs, the iteration log shows the progress of convergence in finding the appropriate set of columns in the reformulated space.

The following information is written to the iteration log:

Iter	indicates the iteration number.
Best Bound	indicates the best dual bound found so far.
Master Objective	indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
Gap	indicates the relative difference between the master objective and the best known dual bound. This indicates how close the algorithm is to convergence. If the relative gap is greater than 1000%, then the absolute gap is written.

CPU Time	indicates the CPU time elapsed (in seconds).
Real Time	indicates the real time elapsed (in seconds).

Entries are made in the log at a frequency that is specified in the LOGFREQ= option. If LOGFREQ=0, then the iteration log is disabled. If the LOGFREQ= value is positive, then an entry is made in the log at the first iteration, at the last iteration, and at intervals that are specified by the LOGFREQ= value. An entry is also made each time an improved bound is found.

The behavior of objective values in the iteration log depends on both the current phase and on which solver you choose. In phase I, the master formulation has an artificial objective value that decreases to 0 when a feasible solution is found. In phase II, the decomposition algorithm maintains a primal feasible solution, so a minimization problem has decreasing objective values in the iteration log.

When you specify LOGLEVEL=MODERATE or LOGLEVEL=AGGRESSIVE in the DECOMP statement, information about the subproblem solves is written before each iteration line.

Iteration Log for MILPs

When the decomposition algorithm solves MILPs, the iteration log shows the progress of convergence in finding the appropriate set of columns in the reformulated space, in addition to the global convergence of the branch-and-bound algorithm for finding an optimal integer solution.

You can control the amount of information at each node by using the LOGLEVEL= option in the DECOMP statement. By default, the continuous iteration log for the root node is written at the interval specified in the LOGFREQ= option in the DECOMP statement. Then the branch-and-bound node log is written at the interval specified in the LOGFREQ= main solver option.

When the algorithm solves MILPs, the continuous iteration log is similar to the iteration log described in the section "Iteration Log for LPs" on page 747 except that information about integer-feasible solutions is also displayed. The following information is printed in the continuous iteration log when the algorithm solves MILPs:

Iter	indicates the iteration number.
Best Bound	indicates the best dual bound found so far.
Master Objective	indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
Best Integer	indicates the objective of the best integer-feasible solution found so far.
LP Gap	indicates the relative difference between the master objective and the best known dual bound. This indicates how close the algorithm for this particular node is to convergence. If the relative gap is greater than 1000%, then the absolute gap is displayed.
IP Gap	indicates the relative difference between the best integer and the best known dual bound. This indicates how close the branch-and-bound algorithm is to convergence. If the relative gap is greater than 1000%, then the absolute gap is displayed.
CPU Time	indicates the CPU time elapsed (in seconds).
Real Time	indicates the real time elapsed (in seconds).

After the root node is complete, the algorithm then moves into the branch-and-bound phase. By default, it displays the branch-and-bound node log and suppresses the continuous iteration log.

The following information is printed in the branch-and-bound node log when the algorithm solves MILPs:

Node	indicates the sequence number of the current node in the search tree.
Active	indicates the current number of active nodes in the branch-and-bound tree.
Sols	indicates the number of feasible solutions found so far.
Best Integer	indicates the objective of the best integer-feasible solution found so far.
Best Bound	indicates the best dual bound found so far.
Gap	indicates the relative difference between the best integer and the best known dual bound. This indicates how close the branch-and-bound algorithm is to convergence. If the relative gap is greater than 1000%, then the absolute gap is displayed.
CPU Time	indicates the CPU time elapsed (in seconds).
Real Time	indicates the real time elapsed (in seconds).

If the LOGLEVEL= option in the DECOMP statement is set to BASIC, MODERATE or AGGRESSIVE, then the continuous iteration log is displayed for each branch-and-bound node at the interval specified in the LOGFREQ= option in the DECOMP statement.

Additional information can be displayed to the log by specifying the LOGLEVEL= option in each of the algorithmic component statements (DECOMP_MASTER, DECOMP_MASTER_IP, and DECOMP_SUBPROB). By default, the individual component log levels are all disabled.

Examples: Decomposition Algorithm

Example 15.1: Multicommodity Flow Problem and METHOD=NETWORK

This example demonstrates how to use the decomposition algorithm to find a minimum-cost multicommodity flow (MMCF) in a directed network. This type of problem was motivation for the development of the original Dantzig-Wolfe decomposition method (Dantzig and Wolfe 1960).

Let $G = (N, A)$ be a directed graph, and let K be a set of commodities. For each link $(i, j) \in A$ and each commodity k, associate a cost per unit of flow, designated by c_{ij}^k. The demand (or supply) at each node $i \in N$ for commodity k is designated as b_i^k, where $b_i^k \geq 0$ denotes a supply node and $b_i^k < 0$ denotes a demand node. Define decision variables x_{ij}^k that denote the amount of commodity k sent from node i and node j. The amount of total flow, across all commodities, that can be sent across each link is bounded above by u_{ij}.

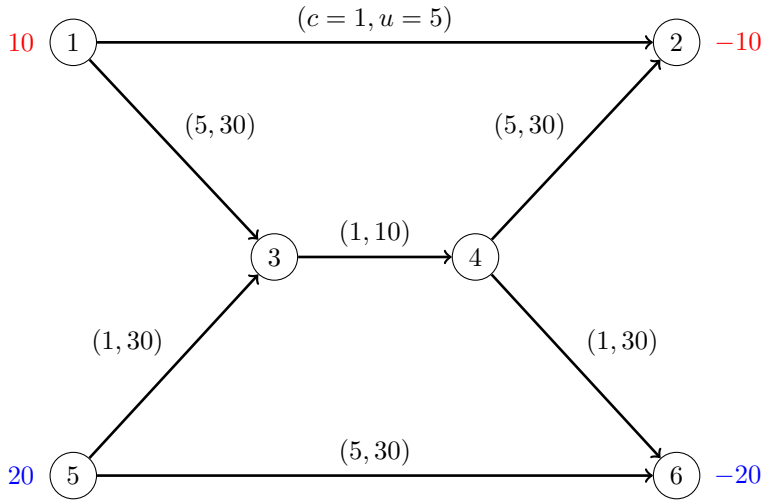
The problem can be modeled as a linear programming problem as follows:

$$\begin{aligned}
&\text{minimize} && \sum_{(i,j)\in A}\sum_{k\in K} c_{ij}^k x_{ij}^k && && \\
&\text{subject to} && \sum_{k\in K} x_{ij}^k \le u_{ij} && (i,j)\in A && \text{(Capacity)}\\
& && \sum_{(i,j)\in A} x_{ij}^k - \sum_{(j,i)\in A} x_{ji}^k = b_i^k && i\in N,\ k\in K && \text{(Balance)}\\
& && x_{ij}^k \ge 0 && (i,j)\in A,\ k\in K &&
\end{aligned}$$

In this formulation, The Capacity constraints limit the total flow across all commodities on each arc. The Balance constraints ensure that the flow of commodities leaving each supply node and entering each demand node are balanced.

Consider the directed graph in Figure 15.5 which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 15.5 Example Network with Two Commodities



The goal in this example is to minimize the total cost of sending two commodities across the network while satisfying all supplies and demands and respecting arc capacities. If there were no arc capacities linking the two commodities, you could solve a separate minimum-cost network flow problem for each commodity one at a time.

The following data set arc_comm_data provides the cost c_{ij}^k of sending a unit of commodity k along arc (i, j):

```
data arc_comm_data;
   input k i j cost;
   datalines;
1 1 2 1
1 1 3 5
1 5 3 1
1 5 6 5
1 3 4 1
1 4 2 5
1 4 6 1
2 1 2 1
2 1 3 5
2 5 3 1
2 5 6 5
2 3 4 1
2 4 2 5
2 4 6 1
;
```

Next, the data set arc_data provides the capacity u_{ij} for each arc:

```
data arc_data;
   input i j capacity;
   datalines;
1 2  5
1 3 30
5 3 30
5 6 30
3 4 10
4 2 30
4 6 30
;

data supply_data;
   input k i supply;
   datalines;
1 1  10
1 2 -10
2 5  20
2 6 -20
;
```

The following PROC OPTMODEL statements find the minimum-cost multicommodity flow:

```
proc optmodel;
   set <num,num,num> ARC_COMM;
   num cost {ARC_COMM};
   read data arc_comm_data into ARC_COMM=[i j k] cost;

   set ARCS = setof {<i,j,k> in ARC_COMM} <i,j>;
   set COMMODITIES = setof {<i,j,k> in ARC_COMM} k;
   set NODES = union {<i,j> in ARCS} {i,j};

   num arcCapacity {ARCS};
   read data arc_data into [i j] arcCapacity=capacity;

   num supply {NODES, COMMODITIES} init 0;
   read data supply_data into [i k] supply;

   var Flow {<i,j,k> in ARC_COMM} >= 0;
   min TotalCost =
      sum {<i,j,k> in ARC_COMM} cost[i,j,k] * Flow[i,j,k];
   con Balance {i in NODES, k in COMMODITIES}:
      sum {<(i),j,(k)> in ARC_COMM} Flow[i,j,k]
       - sum {<j,(i),(k)> in ARC_COMM} Flow[j,i,k] = supply[i,k];
   con Capacity {<i,j> in ARCS}:
      sum {<(i),(j),k> in ARC_COMM} Flow[i,j,k] <= arcCapacity[i,j];
```

Because each Balance constraint involves variables for only one commodity, a decomposition by commodity is a natural choice. In both the OPTLP and OPTMILP procedures, the block identifiers must be consecutive integers starting from 0. In PROC OPTMODEL, the block identifiers only need to be numeric. The following **FOR** loop populates the **.block** constraint suffix with block identifier *k* for commodity *k*:

```
   for{i in NODES, k in COMMODITIES}
      Balance[i,k].block = k;
```

The **.block** constraint suffix for the linking Capacity constraints is left missing, so these constraints become part of the master problem.

The following SOLVE statement uses the DECOMP option to invoke the decomposition algorithm:

```
   solve with LP / presolver=none decomp subprob=(algorithm=nspure);
   print Flow;
quit;
```

Here, the PRESOLVER=NONE option is used, because otherwise the presolver solves this small instance without invoking any solver. Because each subproblem is a pure network flow problem, you can use the ALGORITHM=NSPURE option in the SUBPROB= option to request that a network simplex algorithm for pure networks be used instead of the default algorithm, which for linear programming subproblems is primal simplex.

It turns out for this example that if you specify METHOD=NETWORK (instead of the default METHOD=USER) in the DECOMP option, the network extractor finds the same blocks, one per commodity. To invoke the METHOD=NETWORK option, simply change the SOLVE statement as follows:

```
solve with LP / presolver=none decomp=(method=network);
```

In this case, the default subproblem solver is NSPURE.

The optimal solution and solution summary are displayed in Output 15.1.1.

Output 15.1.1 Solution Summary and Optimal Solution

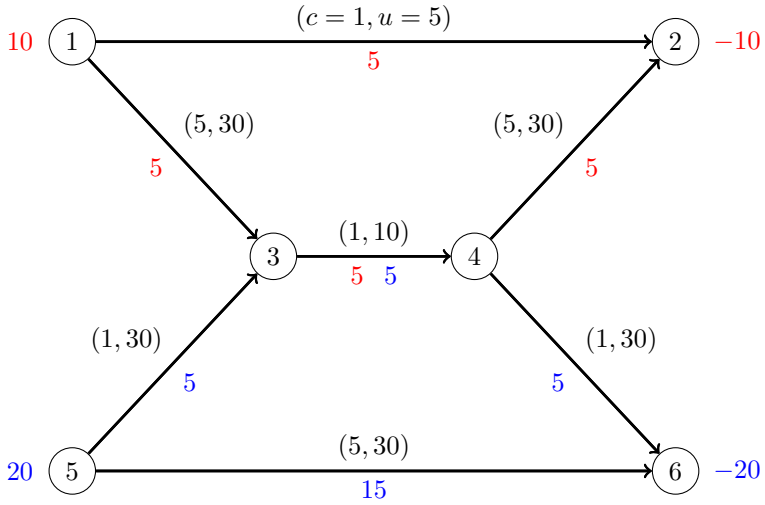
The OPTMODEL Procedure

Solution Summary	
Solver	LP
Algorithm	Decomposition
Objective Function	TotalCost
Solution Status	Optimal
Objective Value	150
Primal Infeasibility	0
Dual Infeasibility	0
Bound Infeasibility	0
Iterations	3
Presolve Time	0.00
Solution Time	0.00

[1]	[2]	[3]	Flow
1	**2**	**1**	5
1	**2**	**2**	0
1	**3**	**1**	5
1	**3**	**2**	0
3	**4**	**1**	5
3	**4**	**2**	5
4	**2**	**1**	5
4	**2**	**2**	0
4	**6**	**1**	0
4	**6**	**2**	5
5	**3**	**1**	0
5	**3**	**2**	5
5	**6**	**1**	0
5	**6**	**2**	15

The optimal solution is shown on the network in Figure 15.6.

Figure 15.6 Optimal Flow on Network with Two Commodities



The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 15.1.2.

Output 15.1.2 Log

```
NOTE: There were 14 observations read from the data set WORK.ARC_COMM_DATA.
NOTE: There were 7 observations read from the data set WORK.ARC_DATA.
NOTE: There were 4 observations read from the data set WORK.SUPPLY_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 14 variables (0 free, 0 fixed).
NOTE: The problem has 19 linear constraints (7 LE, 12 EQ, 0 GE, 0 range).
NOTE: The problem has 42 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The number of block threads has been reduced to 2 threads.
NOTE: The problem has a decomposable structure with 2 blocks. The largest block covers 31.58%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 14 (100%) variables and 12 (63.16%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
       Iter          Best        Master       Gap       CPU      Real
                    Bound     Objective                Time      Time
NOTE: Starting phase 1.
          1        0.0000        1.0000  1.00e+00       0.0       0.0
          2        0.0000        0.0000     0.00%       0.0       0.0
NOTE: Starting phase 2.
          3      150.0000      150.0000     0.00%       0.0       0.0
NOTE: The Decomposition algorithm used 2 threads.
NOTE: The Decomposition algorithm time is 0.00 seconds.
NOTE: Optimal.
NOTE: Objective = 150.
```

Example 15.2: Generalized Assignment Problem

The generalized assignment problem (GAP) is that of finding a maximum profit assignment from n tasks to m machines such that each task is assigned to precisely one machine subject to capacity restrictions on the machines. With each possible assignment, associate a binary variable x_{ij}, which, if set to 1, indicates that machine i is assigned to task j. For ease of notation, define two index sets $M = \{1, \ldots, m\}$ and $N = \{1, \ldots, n\}$. A GAP can be formulated as a MILP as follows:

$$\text{maximize} \quad \sum_{i \in M} \sum_{j \in N} p_{ij} x_{ij}$$

$$\begin{aligned}
\text{subject to} \quad & \sum_{i \in M} x_{ij} = 1 && j \in N && \text{(Assignment)} \\
& \sum_{j \in N} w_{ij} x_{ij} \le b_i && i \in M && \text{(Knapsack)} \\
& x_{ij} \in \{0, 1\} && i \in M,\ j \in N
\end{aligned}$$

In this formulation, Assignment constraints ensure that each task is assigned to exactly one machine. Knapsack constraints ensure that for each machine, the capacity restrictions are met.

Consider the following example taken from Koch et al. (2011) with $n = 24$ tasks to be assigned to $m = 8$ machines. The data set profit_data provides the profit for assigning a particular task to a particular machine:

```
%let NumTasks    = 24;
%let NumMachines = 8;

data profit_data;
   input p1-p&NumTasks;
   datalines;
25 23 20 16 19 22 20 16 15 22 15 21 20 23 20 22 19 25 25 24 21 17 23 17
16 19 22 22 19 23 17 24 15 24 18 19 20 24 25 25 19 24 18 21 16 25 15 20
20 18 23 23 23 17 19 16 24 24 17 23 19 22 23 25 23 18 19 24 20 17 23 23
16 16 15 23 15 15 25 22 17 20 19 16 17 17 20 17 17 18 16 18 15 25 22 17
17 23 21 20 24 22 25 17 22 20 16 22 21 23 24 15 22 25 18 19 19 17 22 23
24 21 23 17 21 19 19 17 18 24 15 15 17 18 15 24 19 21 23 24 17 20 16 21
18 21 22 23 22 15 18 15 21 22 15 23 21 25 25 23 20 16 25 17 15 15 18 16
19 24 18 17 21 18 24 25 18 23 21 15 24 23 18 18 23 23 16 20 20 19 25 21
;
```

The data set weight_data provides the amount of resources used by a particular task when assigned to a particular machine:

```
data weight_data;
   input w1-w&NumTasks;
   datalines;
 8 18 22  5 11 11 22 11 17 22 11 20 13 13  7 22 15 22 24  8  8 24 18  8
24 14 11 15 24  8 10 15 19 25  6 13 10 25 19 24 13 12  5 18 10 24  8  5
22 22 21 22 13 16 21  5 25 13 12  9 24  6 22 24 11 21 11 14 12 10 20  6
13  8 19 12 19 18 10 21  5  9 11  9 22  8 12 13  9 25 19 24 22  6 19 14
25 16 13  5 11  8  7  8 25 20 24 20 11  6 10 10  6 22 10 10 13 21  5 19
19 19  5 11 22 24 18 11  6 13 24 24 22  6 22  5 14  6 16 11  6  8 18 10
24 10  9 10  6 15  7 13 20  8  7  9 24  9 21  9 11 19 10  5 23 20  5 21
 6  9  9  5 12 10 16 15 19 18 20 18 16 21 11 12 22 16 21 25  7 14 16 10
;
```

Finally, the data set capacity_data provides the resource capacity for each machine:

```
data capacity_data;
   input b @@;
   datalines;
36 35 38 34 32 34 31 34
;
```

The following PROC OPTMODEL statements read in the data and define the necessary sets and parameters:

```
proc optmodel;
   /* declare index sets */
   set TASKS    = 1..&NumTasks;
   set MACHINES = 1..&NumMachines;

   /* declare parameters */
   num profit   {MACHINES, TASKS};
   num weight   {MACHINES, TASKS};
   num capacity {MACHINES};

   /* read data sets to populate data */
   read data profit_data   into [i=_n_] {j in TASKS} <profit[i,j]=col('p'||j)>;
   read data weight_data   into [i=_n_] {j in TASKS} <weight[i,j]=col('w'||j)>;
   read data capacity_data into [_n_] capacity=b;
```

The following statements declare the optimization model:

```
   /* declare decision variables */
   var Assign {MACHINES, TASKS} binary;

   /* declare objective */
   max TotalProfit =
      sum {i in MACHINES, j in TASKS} profit[i,j] * Assign[i,j];

   /* declare constraints */
   con Assignment {j in TASKS}:
      sum {i in MACHINES} Assign[i,j] = 1;

   con Knapsack {i in MACHINES}:
      sum {j in TASKS} weight[i,j] * Assign[i,j] <= capacity[i];
```

The following statements use two different decompositions to solve the problem. The first decomposition defines each Assignment constraint as a block and uses the pure network simplex solver for the subproblem. The second decomposition defines each Knapsack constraint as a block and uses the MILP solver for the subproblem.

```
   /* each Assignment constraint defines a block */
   for{j in TASKS}
      Assignment[j].block = j;

   solve with milp / logfreq=1000
      decomp        =()
      decomp_subprob=(algorithm=nspure);

   /* each Knapsack constraint defines a block */
   for{j in TASKS}
      Assignment[j].block = .;
   for{i in MACHINES}
      Knapsack[i].block = i;

   solve with milp / decomp;
quit;
```

The solution summaries are displayed in Output 15.2.1.

Output 15.2.1 Solution Summaries

The OPTMODEL Procedure

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	TotalProfit
Solution Status	Optimal
Objective Value	563
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	0
Best Bound	563
Nodes	1329
Iterations	3896
Presolve Time	0.00
Solution Time	5.96

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	TotalProfit
Solution Status	Optimal
Objective Value	563
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	2.220446E-16
Bound Infeasibility	0
Integer Infeasibility	2.220446E-16
Best Bound	563
Nodes	3
Iterations	57
Presolve Time	0.00
Solution Time	0.33

The iteration log for both decompositions is shown in Output 15.2.2. This example is interesting because it shows the tradeoff between the strength of the relaxation and the difficulty of its resolution. In the first decomposition, the subproblems are totally unimodular and can be solved trivially. Consequently, each iteration of the decomposition algorithm is very fast. However, the bound obtained is as weak as the bound found in direct methods (the LP bound). The weaker bound leads to the need to enumerate more nodes overall. Alternatively, in the second decomposition, the subproblem is the knapsack problem, which is solved using MILP. In this case, the bound is much tighter and the problem solves in very few nodes. The tradeoff, of course, is that each iteration takes longer because solving the knapsack problem is not trivial. Another interesting aspect of this problem is that the subproblem coverage in the second decomposition is much smaller than that of the first decomposition. However, when dealing with MILP, it is not always the size of the coverage that determines the overall effectiveness of a particular choice of decomposition.

Output 15.2.2 Log

```
NOTE: There were 8 observations read from the data set WORK.PROFIT_DATA.
NOTE: There were 8 observations read from the data set WORK.WEIGHT_DATA.
NOTE: There were 8 observations read from the data set WORK.CAPACITY_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 192 variables (0 free, 0 fixed).
NOTE: The problem has 192 binary and 0 integer variables.
NOTE: The problem has 32 linear constraints (8 LE, 24 EQ, 0 GE, 0 range).
NOTE: The problem has 384 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 192 variables, 32 constraints, and 384 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
WARNING: The subproblem solver chosen is an LP solver but at least one block has integer
         variables.
NOTE: The problem has a decomposable structure with 24 blocks. The largest block covers 3.125%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 192 (100%) variables and 24 (75%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best       Master          Best        LP        IP  CPU Real
                   Bound    Objective       Integer       Gap       Gap Time Time
NOTE: Starting phase 1.
         2        0.0000       0.0000             .    0.00%         .    0    0
NOTE: Starting phase 2.
         3      573.5068     567.2553      560.0000    1.09%     2.36%    0    0
         4      569.6460     568.3388      560.0000    0.23%     1.69%    0    0
         5      568.6464     568.6464      560.0000    0.00%     1.52%    0    0
NOTE: Starting branch and bound.
         Node  Active   Sols          Best          Best       Gap     CPU   Real
                                    Integer         Bound             Time   Time
            0       1      1      560.0000      568.6464     1.52%       0      0
         1000     178      3      563.0000      564.2784     0.23%       4      4
         1328       0      3      563.0000      563.0000     0.00%       5      5
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 5.96 seconds.
NOTE: Optimal.
NOTE: Objective = 563.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 192 variables, 32 constraints, and 384 constraint coefficients.
NOTE: The MILP solver is called.
```

Output 15.2.2 *continued*

```
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The problem has a decomposable structure with 8 blocks. The largest block covers 3.125%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 192 (100%) variables and 8 (25%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best       Master          Best        LP        IP  CPU Real
                   Bound    Objective       Integer       Gap       Gap Time Time
NOTE: Starting phase 1.
         2        0.0000       0.0000             .    0.00%         .    0    0
NOTE: Starting phase 2.
         3     820.0000     560.0000     560.0000   31.71%   31.71%    0    0
         8     785.0000     560.0000     560.0000   28.66%   28.66%    0    0
         9     694.7813     560.0000     560.0000   19.40%   19.40%    0    0
         .     694.7813     560.0000     560.0000   19.40%   19.40%    0    0
        10     643.0833     560.0000     560.0000   12.92%   12.92%    0    0
        11     629.5714     560.0000     560.0000   11.05%   11.05%    0    0
        12     607.0000     560.0000     560.0000    7.74%    7.74%    0    0
        13     589.4375     560.0000     560.0000    4.99%    4.99%    0    0
        15     588.1789     561.2947     560.0000    4.57%    4.79%    0    0
        16     583.0000     562.5000     560.0000    3.52%    3.95%    0    0
        17     579.5000     564.0000     560.0000    2.67%    3.36%    0    0
        18     573.0000     564.0000     560.0000    1.57%    2.27%    0    0
        19     572.5000     564.0000     560.0000    1.48%    2.18%    0    0
         .     572.5000     564.0000     560.0000    1.48%    2.18%    0    0
        20     572.5000     564.0000     560.0000    1.48%    2.18%    0    0
        21     566.0000     564.0000     560.0000    0.35%    1.06%    0    0
        22     564.0000     564.0000     560.0000    0.00%    0.71%    0    0
NOTE: Starting branch and bound.
         Node  Active   Sols         Best         Best      Gap     CPU   Real
                                  Integer        Bound             Time   Time
            0       1      1     560.0000     564.0000    0.71%       0      0
            1       1      2     563.0000     564.0000    0.18%       0      0
            2       0      2     563.0000     563.0000    0.00%       0      0
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 0.33 seconds.
NOTE: Optimal.
NOTE: Objective = 563.
```

Example 15.3: Block-Diagonal Structure and METHOD=CONCOMP in Single-Machine Mode

This example demonstrates how you can use the METHOD=CONCOMP option in the DECOMP statement to execute the decomposition algorithm in single-machine mode.

Consider a mixed integer linear program that is defined by the MPS data set mpsdata. In this case, the structure of the model is unknown and only the MPS data set is provided to you.

The following PROC OPTMILP statements solve the problem by using standard methods:

```
proc optmilp
   data = mpsdata;
run;
```

The solution summary is shown in Output 15.3.1.

Output 15.3.1 Solution Summary

The OPTMILP Procedure

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	R0001298
Solution Status	Optimal
Objective Value	120
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	6.039615E-14
Bound Infeasibility	3.019808E-14
Integer Infeasibility	8.437695E-15
Best Bound	120
Nodes	1
Iterations	40025
Presolve Time	0.02
Solution Time	2.33

The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 15.3.2.

Output 15.3.2 Log

```
NOTE: The problem MPSDATA has 388 variables (36 binary, 0 integer, 1 free, 0 fixed).
NOTE: The problem has 1297 constraints (630 LE, 37 EQ, 630 GE, 0 range).
NOTE: The problem has 4204 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 37 variables and 37 constraints.
NOTE: The MILP presolver removed 424 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 351 variables, 1260 constraints, and 3780 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
NOTE: The problem has a decomposable structure with 4 blocks. The largest block covers 25.08%
      of the constraints in the problem. The DECOMP option with METHOD=CONCOMP is recommended
      for solving problems with this structure.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       1    231.0000000     91.4479396  152.60%       0
             0       1       2    120.0000000    120.0000000    0.00%       2
             0       0       2    120.0000000    120.0000000    0.00%       2
NOTE: The MILP solver added 0 cuts with 0 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 120.
```

A note in the log suggests that you can use the decomposition algorithm because of the structure of the problem. The following PROC OPTMILP statements use the METHOD=CONCOMP option in the DECOMP statement in single-machine mode. The PERFORMANCE statement specifies the number of threads to use.

```
proc optmilp
   data       = mpsdata;
   decomp
     loglevel = 2
     method   = concomp;
   performance
     nthreads = 4;
run;
```

The performance information and solution summary are displayed in Output 15.3.3.

Output 15.3.3 Performance Information and Solution Summary

The OPTMILP Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	R0001298
Solution Status	Optimal
Objective Value	120
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	6.528111E-14
Bound Infeasibility	3.28626E-14
Integer Infeasibility	0
Best Bound	120
Nodes	1
Iterations	1
Presolve Time	0.02
Solution Time	0.89

The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 15.3.4. When you specify NTHREADS=4 in the PERFORMANCE statement in single-machine mode, each block is processed simultaneously on each of four threads.

Output 15.3.4 Log

```
NOTE: The OPTMILP procedure is executing in single-machine mode.
NOTE: The problem MPSDATA has 388 variables (36 binary, 0 integer, 1 free, 0 fixed).
NOTE: The problem has 1297 constraints (630 LE, 37 EQ, 630 GE, 0 range).
NOTE: The problem has 4204 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 37 variables and 37 constraints.
NOTE: The MILP presolver removed 424 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 351 variables, 1260 constraints, and 3780 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The DECOMP method value CONCOMP is applied.
NOTE: The problem has a decomposable structure with 4 blocks. The largest block covers 25.08%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 351 (100%) variables and 1260 (100%) constraints.
NOTE: Block 1 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 2 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 3 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 4 has 87 (24.79%) variables and 312 (24.76%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
NOTE: ----------------------------------------------------------------------------------------
NOTE: Starting to process node 0.
NOTE: ----------------------------------------------------------------------------------------
NOTE: Using a starting solution with objective value 231 to provide initial columns.
NOTE: The initial column pool using the starting solution contains 4 columns.
NOTE: The subproblem solver for 4 blocks at iteration 0 is starting.
NOTE: The subproblem solver for 4 blocks used 0.83 (cpu: 2.64) seconds.
NOTE: The initial column pool after generating initial variables contains 8 columns.
NOTE: The master solver at iteration 1 is starting.
NOTE: The master solver used 0.00 (cpu: 0.00) seconds and 0 iterations.
NOTE: The number of active nodes is 0.
NOTE: The objective value of the best integer feasible solution is 120.0000 and the best bound
      is 120.0000.
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 0.87 seconds.
NOTE: Optimal.
NOTE: Objective = 120.
```

In this case, the solver finds that after presolve, the constraint matrix decomposes into block-diagonal form. That is, all the constraints are covered by subproblem blocks, leaving the set of master constraints empty. Because there are no coupling constraints, the problem decomposes into four completely independent problems. If you specify LOGLEVEL=2 in the DECOMP statement, the log displays the size of each block. The blocks in this case are nicely balanced, allowing parallel execution to be efficient.

Example 15.4: Block-Diagonal Structure and METHOD=CONCOMP in Distributed Mode

This example demonstrates how you can use the METHOD=CONCOMP option in the DECOMP statement to execute the decomposition algorithm in distributed mode.

As in Example 15.3, consider a mixed integer linear program that is defined by the MPS data set mpsdata. In this case, the structure of the model is unknown and only the MPS data set is provided to you.

The following PROC OPTMILP statements use the METHOD=CONCOMP option in distributed mode. The PERFORMANCE statement specifies the numbers of threads and nodes to use.

```
proc optmilp
   data        = mpsdata;
   decomp
     loglevel = 2
     method   = concomp;
   performance
     details
     nthreads = 1
     nodes    = 4;
run;
```

The performance information is displayed in Output 15.4.1.

Output 15.4.1 Performance Information

Performance Information	
Host Node	<< your grid host >>
Execution Mode	Distributed
Number of Compute Nodes	4
Number of Threads per Node	1

The solution summary is displayed in Output 15.4.2.

Output 15.4.2 Solution Summary

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	R0001298
Solution Status	Optimal
Objective Value	120
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	7.105427E-15
Bound Infeasibility	1.776357E-15
Integer Infeasibility	5.551115E-16
Best Bound	120
Nodes	1
Iterations	1
Presolve Time	0.02
Solution Time	0.53

The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 15.4.3. When you specify NODES=4 and NTHREADS=1 in the PERFORMANCE statement in distributed mode, each block is processed simultaneously on each of four grid nodes.

Output 15.4.3 Log

```
NOTE: The problem MPSDATA has 388 variables (36 binary, 0 integer, 1 free, 0 fixed).
NOTE: The problem has 1297 constraints (630 LE, 37 EQ, 630 GE, 0 range).
NOTE: The problem has 4204 constraint coefficients.
NOTE: The OPTMILP procedure is executing in the distributed computing environment with 4 worker
      nodes.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 37 variables and 37 constraints.
NOTE: The MILP presolver removed 424 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 351 variables, 1260 constraints, and 3780 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The DECOMP method value CONCOMP is applied.
NOTE: The problem has a decomposable structure with 4 blocks. The largest block covers 25.08%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 351 (100%) variables and 1260 (100%) constraints.
NOTE: Block 1 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 2 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 3 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 4 has 87 (24.79%) variables and 312 (24.76%) constraints.
NOTE: ----------------------------------------------------------------
NOTE: Starting to process node 0.
NOTE: ----------------------------------------------------------------
NOTE: Using a starting solution with objective value 231 to provide initial columns.
NOTE: The initial column pool using the starting solution contains 4 columns.
NOTE: The subproblem solver for 4 blocks at iteration 0 is starting.
NOTE: The subproblem solver for 4 blocks used 0.47 (cpu: 0.00) seconds.
NOTE: The initial column pool after generating initial variables contains 8 columns.
NOTE: The master solver at iteration 1 is starting.
NOTE: The master solver used 0.00 (cpu: 0.00) seconds and 0 iterations.
NOTE: The number of active nodes is 0.
NOTE: The objective value of the best integer feasible solution is 120.0000 and the best bound
      is 120.0000.
NOTE: The Decomposition algorithm time is 0.48 seconds.
NOTE: Optimal.
NOTE: Objective = 120.
NOTE: The data set WORK.PERFINFO has 4 observations and 3 variables.
```

Example 15.5: Block-Angular Structure and METHOD=AUTO

This example demonstrates how you can use the METHOD=AUTO option in the DECOMP statement to execute the decomposition algorithm.

As in Example 15.3, consider a mixed integer linear program that is defined by the MPS data set mpsdata. In this case, the structure of the model is unknown and only the MPS data set is provided to you.

The following PROC OPTMILP statements attempt to solve the problem by using standard methods and a 60-second time limit:

```
proc optmilp
   maxtime = 60
   data    = mpsdata;
run;
```

The solution summary is shown in Output 15.5.1.

Output 15.5.1 Solution Summary

The OPTMILP Procedure

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	Total_Profit
Solution Status	Time Limit Reached
Best Bound	7288.8630869
Nodes	1
Iterations	35694
Presolve Time	0.39
Solution Time	59.81

The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 15.5.2.

Output 15.5.2 Log

```
NOTE: The problem MPSDATA has 52638 variables (16038 binary, 0 integer, 0 free, 0 fixed).
NOTE: The problem has 3949 constraints (3339 LE, 0 EQ, 610 GE, 0 range).
NOTE: The problem has 148866 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 367 constraints.
NOTE: The MILP presolver removed 6606 constraint coefficients.
NOTE: The MILP presolver modified 6606 constraint coefficients.
NOTE: The presolved problem has 52638 variables, 3582 constraints, and 142260 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       0              .   7342.1209242        .       0
             0       1       0              .   7337.8112270        .       7
             0       1       0              .   7325.4946006        .      15
             0       1       0              .   7318.3100411        .      22
             0       1       0              .   7313.6092376        .      31
             0       1       0              .   7308.6489578        .      38
             0       1       0              .   7303.9359225        .      46
             0       1       0              .   7298.3635513        .      54
             0       1       0              .   7293.0296430        .      56
             0       1       0              .   7288.8630869        .      58
NOTE: The MILP solver added 4241 cuts with 99960 cut coefficients at the root.
NOTE: Real time limit reached.
NOTE: No integer solutions found.
```

Standard MILP techniques struggle to find a feasible solution within the specified time limit. The default decomposition method (METHOD=AUTO) attempts to find a block-angular structure by using the matrix-stretching techniques that are described in Grcar (1990) and Aykanat, Pinar, and Çatalyürek (2004).

```
proc optmilp
   data     = mpsdata;
   decomp
     method = auto;
run;
```

The solution summary is displayed in Output 15.5.3.

Output 15.5.3 Solution Summary

The OPTMILP Procedure

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	Total_Profit
Solution Status	Optimal within Relative Gap
Objective Value	6972.3309347
Relative Gap	8.802996E-10
Absolute Gap	6.1377405E-6
Primal Infeasibility	1.949996E-11
Bound Infeasibility	6.999845E-12
Integer Infeasibility	2.664535E-15
Best Bound	6972.3309408
Nodes	1
Iterations	10
Presolve Time	0.41
Solution Time	43.21

The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 15.5.4.

Output 15.5.4 Log

```
NOTE: The OPTMILP procedure is executing in single-machine mode.
NOTE: The problem MPSDATA has 52638 variables (16038 binary, 0 integer, 0 free, 0 fixed).
NOTE: The problem has 3949 constraints (3339 LE, 0 EQ, 610 GE, 0 range).
NOTE: The problem has 148866 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 367 constraints.
NOTE: The MILP presolver removed 6606 constraint coefficients.
NOTE: The MILP presolver modified 6606 constraint coefficients.
NOTE: The presolved problem has 52638 variables, 3582 constraints, and 142260 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The DECOMP method value AUTO is applied.
NOTE: The automated method will attempt to find block-angular form with 4 blocks.
NOTE: The problem has a decomposable structure with 610 blocks. The largest block covers
      0.2233% of the constraints in the problem.
NOTE: The decomposition subproblems cover 52638 (100%) variables and 3574 (99.78%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter         Best       Master         Best       LP       IP  CPU Real
                  Bound    Objective      Integer      Gap      Gap Time Time
NOTE: Starting phase 1.
         1       0.0000     302.6667            . 3.03e+02        .   17   13
         3       0.0000       0.0000            .    0.00%        .   17   13
NOTE: Starting phase 2.
         .    7963.9930    6357.0113    6357.0113   20.18%   20.18%   22   18
         6    7150.3964    6712.7278    6679.8374    6.12%    6.58%   45   26
         7    6997.7702    6946.3759    6946.3759    0.73%    0.73%   68   32
         8    6997.7702    6965.6818    6965.6818    0.46%    0.46%   87   38
         9    6997.7702    6972.2774    6972.2774    0.36%    0.36%   98   41
         .    6997.7702    6972.3309    6972.3309    0.36%    0.36%   99   42
        10    6972.3309    6972.3309    6972.3309    0.00%    0.00%  100   42
         Node  Active   Sols         Best         Best      Gap    CPU   Real
                                  Integer        Bound            Time   Time
            0       0      7    6972.3309    6972.3309    0.00%    100     42
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 42.68 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 6972.3309347.
```

As stated in the log, the automated method attempts to find a balanced block-angular form that contains four blocks (the same setting is used by default in the NTHREADS= option). The algorithm successfully finds such a decomposition and then further decomposes each block into its weakly connected components, resulting in 610 blocks and 99.78% subproblem coverage.

Example 15.6: Bin Packing Problem

The bin packing problem (BPP) finds the minimum number of capacitated bins that are needed to store a set of products of varying size. Define a set P of products, their sizes s_p, and a set $B = \{1, \ldots, |P|\}$ of candidate bins, each having capacity C. Let x_{pb} be a binary variable that, if set to 1, indicates that product p is assigned to bin b. In addition, let y_b be a binary variable that, if set to 1, indicates that bin b is used.

A BPP can be formulated as a MILP as follows:

$$\begin{aligned}
&\text{minimize} && \sum_{b \in B} y_b && && \\
&\text{subject to} && \sum_{b \in B} x_{pb} = 1 && p \in P && \text{(Assignment)} \\
& && \sum_{p \in P} s_p x_{pb} \leq C y_b && b \in B && \text{(Capacity)} \\
& && x_{pb} \in \{0, 1\} && p \in P,\ b \in B && \\
& && y_b \in \{0, 1\} && b \in B &&
\end{aligned}$$

In this formulation, the Assignment constraints ensure that each product is assigned to exactly one bin. The Capacity constraints ensure that the capacity restrictions are met for each bin. In addition, these constraints enforce the condition that if any product is assigned to bin b, then y_b must be positive.

In this formulation, the bin identifier is arbitrary. For example, in any solution, the assignments to bin 1 can be swapped with the assignments to bin 2 without affecting feasibility or the objective value. Consider a decomposition by bin, where the Assignment constraints form the master problem and the Capacity constraints form identical subproblems. As described in the section "Special Case: Identical Blocks and Ryan-Foster Branching" on page 743, this is a situation in which an aggregate formulation and Ryan-Foster branching can greatly improve performance by reducing symmetry.

Consider a series of University of North Carolina basketball games that are recorded on a DVR. The following data set, dvr, provides the name of each game in the column opponent and the size of that game in gigabytes (GB) as it resides on the DVR in the column size:

```
/* game, size (in GBs) */
data dvr;
   input opponent $ size;
   datalines;
Clemson  1.36
Clemson2 1.97
Duke     2.76
Duke2    2.52
FSU      2.56
FSU2     2.34
GT       1.49
GT2      1.12
IN       1.45
KY       1.42
Loyola   1.42
MD       1.33
MD2      2.71
```

```
Miami    1.22
NCSU     2.52
NCSU2    2.54
UConn    1.25
VA       2.33
VA2      2.48
VT       1.41
Vermont  1.28
WM       1.25
WM2      1.23
Wake     1.61
;
```

The goal is to use the fewest DVDs on which to store the games for safekeeping. Each DVD can hold 4.38GB reforded data. The problem can be formulated as a bin packing problem and solved by using PROC OPTMODEL and the decomposition algorithm. The following PROC OPTMODEL statements read in the data, declare the optimization model, and use the decomposition algorithm to solve it:

```
proc optmodel;
   /* read the product and size data */
   set <str> PRODUCTS;
   num size {PRODUCTS};
   read data dvr into PRODUCTS=[opponent] size;

   /* 4.38 GBs per DVD */
   num binsize = 4.38;

   /* the number of products is a trivial upper bound on the
      number of bins needed */
   num upperbound init card(PRODUCTS);
   set BINS = 1..upperbound;

   /* Assign[p,b] = 1, if product p is assigned to bin b */
   var Assign {PRODUCTS, BINS} binary;
   /* UseBin[b] = 1, if bin b is used */
   var UseBin {BINS} binary;

   /* minimize number of bins used */
   min Objective = sum {b in BINS} UseBin[b];

   /* assign each product to exactly one bin */
   con Assignment {p in PRODUCTS}:
      sum {b in BINS} Assign[p,b] = 1;

   /* Capacity constraint on each bin (and definition of UseBin) */
   con Capacity {b in BINS}:
      sum {p in PRODUCTS} size[p] * Assign[p,b] <= binsize * UseBin[b];

   /* decompose by bin (subproblem is a knapsack problem) */
   for {b in BINS} Capacity[b].block = b;

   /* solve using decomp (aggregate formulation) */
   solve with milp / decomp;
```

The following OPTMODEL statements create a sequential numbering of the bins and then output to the data set dvd the optimal assignments of games to bins:

```
   /* create a map from arbitrary bin number to sequential bin number */
   num binId init 1;
   num binMap {BINS};
   for {b in BINS: UseBin[b].sol > 0.5} do;
      binMap[b] = binId;
      binId     = binId + 1;
   end;

   /* create map of product to bin from solution */
   num bin {PRODUCTS};
   for {p in PRODUCTS} do;
      for {b in BINS: Assign[p,b].sol > 0.5} do;
         bin[p] = binMap[b];
         leave;
      end;
   end;

   /* create solution data */
   create data dvd from [product] bin size;
quit;
```

The solution summary is displayed in Output 15.6.1.

Output 15.6.1 Solution Summary

The OPTMODEL Procedure

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	Objective
Solution Status	Optimal
Objective Value	11
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	0
Best Bound	11
Nodes	1
Iterations	18
Presolve Time	0.02
Solution Time	0.25

The iteration log is displayed in Output 15.6.2.

Output 15.6.2 Log

```
NOTE: There were 24 observations read from the data set WORK.DVR.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 600 variables (0 free, 0 fixed).
NOTE: The problem has 600 binary and 0 integer variables.
NOTE: The problem has 48 linear constraints (24 LE, 24 EQ, 0 GE, 0 range).
NOTE: The problem has 1176 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 600 variables, 48 constraints, and 1176 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: All blocks are identical and the master model is set partitioning.
NOTE: The Decomposition algorithm is using an aggregate formulation and Ryan-Foster branching.
NOTE: The number of block threads has been reduced to 1 threads.
NOTE: The problem has a decomposable structure with 24 blocks. The largest block covers 2.083%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 600 (100%) variables and 24 (50%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best       Master          Best       LP        IP  CPU Real
                   Bound    Objective       Integer      Gap       Gap Time Time
         .         0.0000      11.0000       11.0000 1.10e+01 1.10e+01    0    0
         .         0.0000      11.0000       11.0000 1.10e+01 1.10e+01    0    0
        10         0.0000      11.0000       11.0000 1.10e+01 1.10e+01    0    0
        12         3.0000      11.0000       11.0000  266.67%  266.67%    0    0
        18        11.0000      11.0000       11.0000    0.00%    0.00%    0    0
          Node  Active   Sols         Best         Best      Gap    CPU   Real
                                   Integer        Bound            Time   Time
             0       0      2      11.0000      11.0000    0.00%      0      0
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 0.23 seconds.
NOTE: Optimal.
NOTE: Objective = 11.
NOTE: The data set WORK.DVD has 24 observations and 3 variables.
```

The following call to PROC SORT sorts the assignments by bin:

```
proc sort data=dvd;
   by bin;
run;
```

The optimal assignments from the output data set dvd are displayed in Figure 15.7.

Figure 15.7 Optimal Assignment of Games to DVDs

bin=1

product	size
WM	1.25
WM2	1.23
Wake	1.61
bin	**4.09**

bin=2

product	size
Duke2	2.52
UConn	1.25
bin	**3.77**

bin=3

product	size
Miami	1.22
VA2	2.48
bin	**3.70**

bin=4

product	size
MD	1.33
VA	2.33
bin	**3.66**

bin=5

product	size
Loyola	1.42
NCSU2	2.54
bin	**3.96**

bin=6

product	size
KY	1.42
NCSU	2.52
bin	**3.94**

Figure 15.7 *continued*

bin=7

product	size
IN	1.45
MD2	2.71
bin	**4.16**

bin=8

product	size
GT	1.49
VT	1.41
Vermont	1.28
bin	**4.18**

bin=9

product	size
Clemson2	1.97
FSU2	2.34
bin	**4.31**

bin=10

product	size
Clemson	1.36
FSU	2.56
bin	**3.92**

bin=11

product	size
Duke	2.76
GT2	1.12
bin	**3.88**
	43.57

In this example, the objective function ensures that there exists an optimal solution that never assigns a product to more than one bin. Therefore, you could instead model the Assignment constraint as an inequality rather than an equality. In this case, the best performance would come from forcing the use of an aggregate formulation and Ryan-Foster branching by specifying the option VARSEL=RYANFOSTER. An example of doing this is shown in Example 15.8.

Example 15.7: Resource Allocation Problem

This example describes a model for selecting tasks to be run on a shared resource (Gamrath 2010). Consider a set I of tasks and a resource capacity C. Each item $i \in I$ has a profit p_i, a resource utilization level w_i, a starting period s_i, and an ending period e_i. The time horizon that is considered is from the earliest starting time to the latest ending time of all tasks. With each task, associate a binary variable x_i, which, if set to 1, indicates that the task is running from its start time until just before its end time. A task consumes capacity if it is running. The goal is to select which tasks to run in order to maximize profit while not

exceeding the shared resource capacity. Let $S = \{s_i \mid i \in I\}$ define the set of start times for all tasks, and let $L_s = \{i \in I \mid s_i \leq s < e_i\}$ define the set of tasks that are running at each start time $s \in S$. You can model the problem as a mixed integer linear programming problem as follows:

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{i \in I} p_i x_i & \\ \text{subject to} & \displaystyle\sum_{i \in L_s} w_i x_i \leq C & s \in S \qquad \text{(CapacityCon)} \\ & x_i \in \{0,1\} & i \in I \end{array}$$

In this formulation, CapacityCon constraints ensure that the running tasks do not exceed the resource capacity. To illustrate, consider the following five-task example with data: $p_i = (6, 8, 5, 9, 8)$, $w_i = (8, 5, 3, 4, 3)$, $s_i = (1, 3, 5, 7, 8)$, $e_i = (5, 8, 9, 17, 10)$, and $C = 10$. The formulation leads to a constraint matrix that has a *staircase structure* that is determined by tasks coming online and offline:

$$\begin{array}{lrcrcrcrcl} \text{maximize} & 6x_1 & + & 8x_2 & + & 5x_3 & + & 9x_4 & + & 8x_5 & \\ \text{subject to} & 8x_1 & & & & & & & & & \leq 10 \\ & 8x_1 & + & 5x_2 & & & & & & & \leq 10 \\ & & & 5x_2 & + & 3x_3 & & & & & \leq 10 \\ & & & 5x_2 & + & 3x_3 & + & 4x_4 & & & \leq 10 \\ & & & & + & 3x_3 & + & 4x_4 & + & 3x_5 & \leq 10 \\ & x_i \in \{0,1\} & & i \in I & & & & & & & \end{array}$$

Lagrangian Decomposition

This formulation clearly has no decomposable structure. However, you can use a common modeling technique known as *Lagrangian decomposition* to bring the model into block-angular form. Lagrangian decomposition works by first partitioning the constraints into blocks. Then, each original variable is split into multiple copies of itself, one copy for each block in which the variable has a nonzero coefficient in the constraint matrix. Constraints are added to enforce the equality of each copy of the original variable. Then, you can write the original constraints in block-angular form by using the duplicate variables.

To apply Lagrangian decomposition to the resource allocation problem, define a set B of blocks and let S_b define the set of start times for a given block b, such that $S = \cup_{b \in B} S_b$. Given this partition of start times, let B_i define the set of blocks in which task $i \in I$ is scheduled to be running. Now, for each task $i \in I$, define duplicate variables x_i^b for each $b \in B_i$. Let m_i define the minimum block index for each class of variable that represents task i. You can now model the problem in block-angular form as follows:

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{i \in I} p_i x_i^{m_i} & \\ \text{subject to} & x_i^b = x_i^{m_i} & i \in I,\ b \in B_i \setminus \{m_i\} \qquad \text{(LinkDupVarsCon)} \\ & \displaystyle\sum_{i \in L_s} w_i x_i^b \leq C & b \in B,\ s \in S_b \qquad \text{(CapacityCon)} \\ & x_i^b \in \{0,1\} & i \in I,\ b \in B_i \end{array}$$

In this formulation, the LinkDupVarsCon constraints ensure that the duplicate variables are equal to the original variables. Now, the five-task example has been transformed from a staircase structure to a block-angular structure:

$$
\begin{array}{lrcrcrcrcrcrcrcrcl}
\text{maximize} & 6x_1^1 & + & 8x_2^1 & + & & & 5x_3^2 & + & 9x_4^2 & + & & & & & 8x_5^3 & \\
\text{subject to} & & & x_2^1 & - & x_2^2 & & & & & & & & & & & = \; 0 \\
 & & & & & & & x_3^2 & - & & & x_3^3 & & & & & = \; 0 \\
 & & & & & & & & & x_4^2 & & & - & x_4^3 & & & = \; 0 \\
 & 8x_1^1 & & & & & & & & & & & & & & & \leq \; 10 \\
 & 8x_1^1 & + & 5x_2^1 & & & & & & & & & & & & & \leq \; 10 \\
 & & & & & 5x_2^2 & + & 3x_3^2 & & & & & & & & & \leq \; 10 \\
 & & & & & 5x_2^2 & + & 3x_3^2 & + & 4x_4^2 & & & & & & & \leq \; 10 \\
 & & & & & & & & & & & 3x_3^3 & + & 4x_4^3 & + & 3x_5^3 & \leq \; 10 \\
 & x_i^b \in \{0,1\} \quad i \in I,\ b \in B_i & & & & & & & & & & & & & & &
\end{array}
$$

To see how to apply Lagrangian decomposition in PROC OPTMODEL, consider the data set TaskData from Caprara, Furini, and Malaguti (2010), which consists of $|I| = 2{,}916$ tasks:

```
data TaskData;
   input profit weight start end;
   datalines;
99 92 1 9
56 30 1 3
39 73 1 20
86 76 1 9
...
24 94 768 769
95 40 768 769
66 17 768 769
18 48 768 769
97 23 768 769
;
```

Using the MILP Solver Directly in PROC OPTMODEL

The following PROC OPTMODEL statements read in the data and solve the original staircase formulation by calling the MILP solver directly:

```
%macro SetupData(task_data=, capacity=);
   set TASKS;
   num capacity=&capacity;
   num profit{TASKS}, weight{TASKS}, start{TASKS}, end{TASKS};

   read data &task_data into TASKS=[_n_] profit weight start end;
   /* the set of start times */

   set STARTS = setof{i in TASKS} start[i];
   /* the set of tasks i that are active at a given start time s */
   set TASKS_START{s in STARTS}
      = {i in TASKS: start[i] <= s < end[i]};
%mend SetupData;
```

```
%macro ResourceAllocation_Direct(task_data=, capacity=);
   proc optmodel;
      %SetupData(task_data=&task_data,capacity=&capacity);

      /* select task i to come online from period [start to end) */
      var x{TASKS} binary;

      /* maximize the total profit of running tasks */
      max TotalProfit = sum{i in TASKS} profit[i] * x[i];

      /* enforce that the shared resource capacity is not exceeded */
      con CapacityCon{s in STARTS}:
         sum{i in TASKS_START[s]} weight[i] * x[i] <= capacity;

      solve with milp / maxtime=200 logfreq=10000;
   quit;
%mend ResourceAllocation_Direct;

%ResourceAllocation_Direct(task_data=TaskData, capacity=100);
```

The problem summary and solution summary are displayed in Output 15.7.1.

Output 15.7.1 Problem Summary and Solution Summary

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Maximization
Objective Function	TotalProfit
Objective Type	Linear
Number of Variables	2916
Bounded Above	0
Bounded Below	0
Bounded Below and Above	2916
Free	0
Fixed	0
Binary	2916
Integer	0
Number of Constraints	768
Linear LE (<=)	768
Linear EQ (=)	0
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	23236

Output 15.7.1 *continued*

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	TotalProfit
Solution Status	Time Limit Reached
Objective Value	40977.001015
Relative Gap	0.0009889128
Absolute Gap	40.562795844
Primal Infeasibility	9.8717581E-7
Bound Infeasibility	9.6E-7
Integer Infeasibility	9.0909091E-6
Best Bound	41017.563811
Nodes	59036
Iterations	2744264
Presolve Time	0.08
Solution Time	199.65

The iteration log, which contains the problem statistics, the progress of the solution, and the best solution found, is shown in Output 15.7.2.

Output 15.7.2 Log

```
NOTE: There were 2916 observations read from the data set WORK.TASKDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2916 variables (0 free, 0 fixed).
NOTE: The problem has 2916 binary and 0 integer variables.
NOTE: The problem has 768 linear constraints (768 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 23236 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The remaining solution time after problem generation and solver initialization is 199.63
      seconds.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 1021 variables and 126 constraints.
NOTE: The MILP presolver removed 12544 constraint coefficients.
NOTE: The MILP presolver modified 242 constraint coefficients.
NOTE: The presolved problem has 1895 variables, 642 constraints, and 10692 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       3  33939.0000000         109181   68.91%       0
             0       1       3  33939.0000000  45941.6685826   26.13%       0
             0       1       7  39422.0000000  44320.4138809   11.05%       0
             0       1       8  39456.0000000  43343.6560980    8.97%       0
             0       1       8  39456.0000000  42715.1306771    7.63%       0
             0       1       8  39456.0000000  42260.0845046    6.64%       1
             0       1       8  39456.0000000  42063.0122702    6.20%       1
             0       1       8  39456.0000000  41907.1478516    5.85%       1
             0       1       8  39456.0000000  41788.4878466    5.58%       1
             0       1      10  40336.0000000  41717.1427070    3.31%       1
             0       1      10  40336.0000000  41670.3860340    3.20%       1
             0       1      10  40336.0000000  41622.6363124    3.09%       1
             0       1      10  40336.0000000  41592.6013404    3.02%       2
             0       1      10  40336.0000000  41575.0372852    2.98%       2
             0       1      10  40336.0000000  41559.1573069    2.94%       2
             0       1      10  40336.0000000  41546.9607676    2.91%       2
             0       1      10  40336.0000000  41531.5214656    2.88%       2
             0       1      10  40336.0000000  41520.9698542    2.85%       2
             0       1      10  40336.0000000  41505.2690251    2.82%       2
             0       1      10  40336.0000000  41493.8507879    2.79%       2
             0       1      10  40336.0000000  41489.0036876    2.78%       2
             0       1      10  40336.0000000  41483.1887377    2.77%       2
             0       1      10  40336.0000000  41479.2555606    2.76%       2
             0       1      10  40336.0000000  41474.6301544    2.75%       2
             0       1      10  40336.0000000  41472.5678907    2.74%       2
             0       1      11  40361.0000000  41472.5678907    2.68%       2
NOTE: The MILP solver added 773 cuts with 8073 cut coefficients at the root.
          3356    2147      12  40724.0009471  41129.1381702    0.99%      15
          4688    2613      13  40838.0005660  41121.0473036    0.69%      17
          6085    2775      14  40860.0006866  41103.8200082    0.59%      20
```

Output 15.7.2 *continued*

7226	3479	15	40898.0007442	41097.0438201	0.48%	23
7371	3459	16	40901.0010278	41096.3496200	0.48%	23
7534	3433	17	40924.0006857	41096.1865079	0.42%	24
8153	3676	18	40926.0006765	41094.4182925	0.41%	24
10000	4652	18	40926.0006765	41084.1626350	0.38%	30
20000	12724	18	40926.0006765	41060.7946651	0.33%	57
21354	13704	19	40935.0007344	41058.8273130	0.30%	61
21630	13325	20	40962.0010285	41058.6734581	0.24%	61
22255	11817	21	40977.0010152	41057.8058161	0.20%	63
30000	12721	21	40977.0010152	41044.6270814	0.16%	91
40000	17764	21	40977.0010152	41031.9691072	0.13%	129
50000	22196	21	40977.0010152	41023.4407885	0.11%	167
59035	25755	21	40977.0010152	41017.5638111	0.10%	199

```
NOTE: Real time limit reached.
NOTE: Objective of the best integer solution found = 40977.001015.
```

Using the Decomposition Algorithm in PROC OPTMODEL

To transform this data into block-angular form, first sort the task data to help reduce the number of duplicate variables that are needed in the reformulation as follows:

```
proc sort data=TaskData;
   by start end;
run;
```

Then, create the partition of constraints into blocks of size **block_size** as follows:

```
%macro ResourceAllocation_Decomp(task_data=, capacity=, block_size=);
   proc optmodel;
      %SetupData(task_data=&task_data,capacity=&capacity);
      /* partition into blocks of size blocks_size */
      num block_size = &block_size;
      num num_blocks = ceil( card(TASKS) / block_size );
      set BLOCKS     = 1..num_blocks;

      /* the set of starts s for which task i is active */
      set STARTS_TASK{i in TASKS} = {s in STARTS: start[i] <= s < end[i]};

      /* partition the start times into blocks of size block_size */
      set STARTS_BLOCK{BLOCKS} init {};
      num block_id    init 1;
      num block_count init 0;
      for{s in STARTS} do;
         STARTS_BLOCK[block_id] = STARTS_BLOCK[block_id] union {s};
         block_count = block_count + 1;
         if(mod(block_count, block_size) = 0) then
             block_id = block_id + 1;
      end;
```

Then, use the following PROC OPTMODEL statements to define the block-angular formulation and solve the problem by using the decomposition algorithm, the PRESOLVER=BASIC option, and **block_size=20**. Because this reformulation is equivalent to the original staircase formulation, disabling some of the advanced presolver techniques ensures that the model maintains block-angularity.

```
      /* blocks in which task i is online */
      set BLOCKS_TASK{i in TASKS} =
         {b in BLOCKS: card(STARTS_BLOCK[b] inter STARTS_TASK[i]) > 0};

      /* minimum block id in which task i is online */
      num min_block{i in TASKS} = min{b in BLOCKS_TASK[i]} b;

      /* select task i to come online from period [start to end)
         in each block */
      var x{i in TASKS, b in BLOCKS_TASK[i]} binary;

      /* maximize the total profit of running tasks */
      max TotalProfit = sum{i in TASKS} profit[i] * x[i,min_block[i]];

      /* enforce that task selection is consistent across blocks */
      con LinkDupVarsCon{i in TASKS, b in BLOCKS_TASK[i] diff {min_block[i]}}:
         x[i,b] = x[i,min_block[i]];

      /* enforce that the shared resource capacity is not exceeded */
      con CapacityCon{b in BLOCKS, s in STARTS_BLOCK[b]}:
         sum{i in TASKS_START[s]} weight[i] * x[i,b] <= capacity;

      /* define blocks for decomposition algorithm */
      for{b in BLOCKS, s in STARTS_BLOCK[b]} CapacityCon[b,s].block = b;

      solve with milp / presolver=basic decomp;
   quit;
%mend ResourceAllocation_Decomp;

%ResourceAllocation_Decomp(task_data=TaskData, capacity=100, block_size=20);
```

The problem summary and solution summary are displayed in Output 15.7.3. Compared to the original formulation, the numbers of variables and constraints are increased by the number of duplicate variables.

Output 15.7.3 Problem Summary and Solution Summary

The OPTMODEL Procedure

Problem Summary	
Objective Sense	Maximization
Objective Function	TotalProfit
Objective Type	Linear
Number of Variables	3924
Bounded Above	0
Bounded Below	0
Bounded Below and Above	3924
Free	0
Fixed	0
Binary	3924
Integer	0
Number of Constraints	1776
Linear LE (<=)	768
Linear EQ (=)	1008
Linear GE (>=)	0
Linear Range	0
Constraint Coefficients	25252

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	TotalProfit
Solution Status	Optimal within Relative Gap
Objective Value	40982
Relative Gap	0.0000935404
Absolute Gap	3.8338293093
Primal Infeasibility	2.842171E-14
Bound Infeasibility	1.998401E-15
Integer Infeasibility	3.552714E-15
Best Bound	40985.833829
Nodes	15
Iterations	486
Presolve Time	0.05
Solution Time	106.67

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 15.7.4.

Output 15.7.4 Log

```
NOTE: There were 2916 observations read from the data set WORK.TASKDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3924 variables (0 free, 0 fixed).
NOTE: The problem has 3924 binary and 0 integer variables.
NOTE: The problem has 1776 linear constraints (768 LE, 1008 EQ, 0 GE, 0 range).
NOTE: The problem has 25252 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value BASIC is applied.
NOTE: The MILP presolver removed 4 variables and 0 constraints.
NOTE: The MILP presolver removed 23 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 3920 variables, 1776 constraints, and 25229 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The problem has a decomposable structure with 39 blocks. The largest block covers 1.126%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 3920 (100%) variables and 768 (43.24%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
```

Iter	Best Bound	Master Objective	Best Integer	LP Gap	IP Gap	CPU Time	Real Time
.	44109.0014	40043.0000	40043.0000	9.22%	9.22%	1	0
.	44109.0014	40043.0000	40043.0000	9.22%	9.22%	8	3
10	44109.0014	40043.0000	40043.0000	9.22%	9.22%	9	3
11	44109.0014	40059.0000	40059.0000	9.18%	9.18%	10	3
12	44109.0014	40071.0000	40071.0000	9.15%	9.15%	11	4
16	44109.0014	40233.0000	40233.0000	8.79%	8.79%	15	5
18	44109.0014	40272.0000	40272.0000	8.70%	8.70%	17	6
.	44109.0014	40310.0000	40310.0000	8.61%	8.61%	19	7
20	44109.0014	40310.0000	40310.0000	8.61%	8.61%	20	7
25	43411.8703	40443.0000	40443.0000	6.84%	6.84%	25	9
28	43411.8703	40463.0000	40463.0000	6.79%	6.79%	28	10
30	43411.8703	40463.0000	40502.0000	6.79%	6.70%	30	11
34	43411.8703	40599.0000	40599.0000	6.48%	6.48%	34	12
37	43411.8703	40635.0000	40635.0000	6.40%	6.40%	37	13
39	43411.8703	40670.0000	40670.0000	6.32%	6.32%	40	14
40	43411.8703	40670.0000	40713.0000	6.32%	6.22%	40	14
44	43411.8703	40752.0000	40752.0000	6.13%	6.13%	45	16
47	43411.8703	40877.0000	40877.0000	5.84%	5.84%	47	17
50	43411.8703	40923.5000	40915.0000	5.73%	5.75%	48	18
52	43411.8703	40934.0000	40934.0000	5.71%	5.71%	50	18
55	43411.8703	40968.7333	40958.0000	5.63%	5.65%	54	20
60	43411.8703	40975.4242	40958.0000	5.61%	5.65%	59	21
61	41130.5002	40987.1333	40958.0000	0.35%	0.42%	60	22
62	41100.3633	40990.5833	40958.0000	0.27%	0.35%	62	22
63	41039.4670	40990.5833	40958.0000	0.12%	0.20%	63	23

Output 15.7.4 *continued*

```
        70   41039.4670   40990.5833   40958.0000     0.12%     0.20%   66   24
         .   41039.4670   40997.5833   40965.0000     0.10%     0.18%   66   24
        80   41039.4670   40997.5833   40965.0000     0.10%     0.18%   66   24
        81   41008.5834   40997.5833   40965.0000     0.03%     0.11%   67   25
        82   40999.0834   40997.5833   40965.0000     0.00%     0.08%   67   25
NOTE: The Decomposition algorithm stopped on the continuous RELOBJGAP= option.
NOTE: Starting branch and bound.
         Node  Active   Sols         Best         Best      Gap    CPU   Real
                                  Integer        Bound            Time   Time
            0       1     55   40965.0000   40999.0834    0.08%     67     25
           10      12     56   40977.0000   40986.8336    0.02%    309    101
           11       9     57   40982.0000   40986.8336    0.01%    313    102
           14       6     57   40982.0000   40985.8338    0.01%    323    106
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 106.55 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 40982.
```

The Trade-Off between Coverage and Subproblem Difficulty

The reformulation of this resource allocation problem provides a nice example of the potential trade-offs in modeling a problem for use with the decomposition algorithm. As seen in Example 15.2, the strength of the bound is an important factor in the overall performance of the algorithm, but it is not always correlated to the magnitude of the subproblem coverage. In the current example, the block size determines the number of blocks. Moreover, it determines the number of linking variables that are needed in the reformulation. At one extreme, if the block size is set to be $|S|$, then the number of blocks is 1, and the number of copies of original variables is 0. Using one block would be equivalent to the original staircase formulation and would not yield a model conducive to decomposition. As the number of blocks is increased, the number of linking variables increases (the size of the master problem), the strength of the decomposition bound decreases, and the difficulty of solving the subproblems decreases. In addition, as the number of blocks and their relative difficulty change, the efficient utilization of your machine's parallel architecture can be affected.

The previous section used a block size of 20. The following statement calls the decomposition algorithm and uses a block size of 80:

```
%ResourceAllocation_Decomp(task_data=TaskData, capacity=100, block_size=80);
```

The solution summary is displayed in Output 15.7.5.

Output 15.7.5 Solution Summary

The OPTMODEL Procedure

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	TotalProfit
Solution Status	Optimal within Relative Gap
Objective Value	40982
Relative Gap	1.8300718E-9
Absolute Gap	0.000075
Primal Infeasibility	0
Bound Infeasibility	4.662937E-15
Integer Infeasibility	8.770762E-15
Best Bound	40982.000075
Nodes	1
Iterations	41
Presolve Time	0.03
Solution Time	51.03

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 15.7.6.

This version of the model provides a stronger initial bound and solves to optimality in the root node.

Output 15.7.6 Log

```
NOTE: There were 2916 observations read from the data set WORK.TASKDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3151 variables (0 free, 0 fixed).
NOTE: The problem has 3151 binary and 0 integer variables.
NOTE: The problem has 1003 linear constraints (768 LE, 235 EQ, 0 GE, 0 range).
NOTE: The problem has 23706 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value BASIC is applied.
NOTE: The MILP presolver removed 5 variables and 0 constraints.
NOTE: The MILP presolver removed 29 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 3146 variables, 1003 constraints, and 23677 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The problem has a decomposable structure with 10 blocks. The largest block covers 7.976%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 3146 (100%) variables and 768 (76.57%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best       Master          Best       LP       IP  CPU Real
                  Bound    Objective       Integer      Gap      Gap Time Time
         .   41761.9995   40069.0000   40069.0000    4.05%    4.05%    5    2
         5   41761.9995   40112.0000   40112.0000    3.95%    3.95%   16    6
         8   41761.9995   40116.0000   40116.0000    3.94%    3.94%   22    9
         .   41761.9995   40334.0000   40334.0000    3.42%    3.42%   23    9
        10   41761.9995   40334.0000   40334.0000    3.42%    3.42%   26   10
        16   41761.9995   40498.0000   40498.0000    3.03%    3.03%   55   25
         .   41761.9995   40722.0000   40722.0000    2.49%    2.49%   65   30
        20   41604.9657   40722.0000   40722.0000    2.12%    2.12%   68   31
        22   41604.9657   40813.0000   40813.0000    1.90%    1.90%   75   34
        25   41477.7513   40813.0000   40813.0000    1.60%    1.60%   88   39
        27   41212.1672   40884.0000   40884.0000    0.80%    0.80%   90   40
        29   41212.1672   40937.0000   40937.0000    0.67%    0.67%   93   41
        30   41212.1672   40937.0000   40937.0000    0.67%    0.67%   96   42
        34   41212.1672   40961.0000   40961.0000    0.61%    0.61%  104   45
        36   41212.1672   40982.0000   40982.0000    0.56%    0.56%  108   47
         .   41212.1672   40982.0000   40982.0000    0.56%    0.56%  113   49
        40   41212.1672   40982.0000   40982.0000    0.56%    0.56%  115   49
        41   40982.0001   40982.0000   40982.0000    0.00%    0.00%  117   50
         Node  Active   Sols         Best         Best      Gap    CPU   Real
                                  Integer        Bound             Time   Time
            0       0     37   40982.0000   40982.0001    0.00%    117     50
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 50.93 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 40982.
```

Example 15.8: Vehicle Routing Problem

The vehicle routing problem (VRP) finds a minimum-cost routing of a fixed number of vehicles to service the demands of a set of customers. Define a set $C = \{2, \ldots, |C| + 1\}$ of customers, and a demand, d_c, for each customer c. Let $N = C \cup \{1\}$ be the set of nodes, including the vehicle depot, which are designated as node $i = 1$. Let $A = N \times N$ be the set of arcs, V be the set of vehicles (each of which has capacity L), and c_{ij} be the travel time from node i to node j.

Let y_{ik} be a binary variable that, if set to 1, indicates that node i is visited by vehicle k. Let z_{ijk} be a binary variable that, if set to 1, indicates that arc (i, j) is traversed by vehicle k, and let x_{ijk} be a continuous variable that denotes the amount of product (flow) on arc (i, j) that is carried by vehicle k.

A VRP can be formulated as a MILP as follows:

$$\text{minimize} \quad \sum_{(i,j)\in A} \sum_{k\in V} c_{ij} z_{ijk}$$

$$\begin{aligned}
\text{subject to} \quad \sum_{k\in V} y_{ik} &\geq 1 && i \in C && \text{(Assignment)} \\
\sum_{(i,j)\in A} z_{ijk} &= y_{ik} && i \in N,\ k \in V && \text{(LeaveNode)} \\
\sum_{(j,i)\in A} z_{jik} &= y_{ik} && i \in N,\ k \in V && \text{(EnterNode)} \\
\sum_{(j,i)\in A} x_{jik} - \sum_{(i,j)\in A} x_{ijk} &= d_i y_{ik} && i \in C,\ k \in V && \text{(FlowBalance)} \\
x_{ijk} &\leq L z_{ijk} && (i,j) \in A,\ k \in V && \text{(VehicleCapacity)} \\
y_{1k} &= 1 && k \in V && \text{(Depot)} \\
x_{ijk} &\geq 0 && (i,j) \in A,\ k \in V \\
y_{ik} &\in \{0,1\} && i \in N,\ k \in V \\
z_{ijk} &\in \{0,1\} && (i,j) \in A,\ k \in V
\end{aligned}$$

In this formulation, the Assignment constraints ensure that each customer is serviced by at least one vehicle. The objective function ensures that there exists an optimal solution that never assigns a customer to more than one vehicle. The LeaveNode and EnterNode constraints enforce the condition that if node i is visited by vehicle k, then vehicle k must use exactly one arc that enters node i and one arc that leaves node i. Conversely, if node i is not visited by vehicle k, then no arcs that enter or leave node i can be used by vehicle k. The FlowBalance constraints define flow conservation at each node for each vehicle. That is, if a node i is visited by vehicle k, then the amount of product from vehicle k that enters and leaves that node must equal the demand at that node. Conversely, if node i is not visited by vehicle k, then the amount of product from vehicle k that enters and leaves that node must be 0. The VehicleCapacity constraints enforce the condition that the amount of product in each vehicle must always be less than or equal to the vehicle capacity L. Finally, the Depot constraints enforce the condition that each vehicle must start and end at the depot node.

In this formulation, the vehicle identifier is arbitrary. Consider a decomposition by vehicle, where the Assignment constraints form the master problem and all other constraints form identical routing subproblems. As described in the section "Special Case: Identical Blocks and Ryan-Foster Branching" on page 743, this is a situation in which an aggregate formulation can greatly improve performance by reducing symmetry.

Because you know that there exists an optimal solution that satisfies the master Assignment constraints at equality, you can force the use of Ryan-Foster branching by specifying the option VARSEL=RYANFOSTER.

VRPLIB, located at http://www.coin-or.org/SYMPHONY/branchandcut/VRP/data/index.htm, is a set of benchmark instances of the VRP. The following data set, vrpdata, represents an instance from VRPLIB that has 22 nodes and eight vehicles (P-n22-k8.vrp), which was originally described in Augerat et al. (1995). The data set lists each node, its coordinates, and its demand.

```
/* number of vehicles available */
%let num_vehicles = 8;
/* capacity of each vehicle */
%let capacity = 3000;
/* node, x coordinate, y coordinate, demand */
data vrpdata;
   input node x y demand;
   datalines;
1  145 215    0
2  151 264 1100
3  159 261  700
4  130 254  800
5  128 252 1400
6  163 247 2100
7  146 246  400
8  161 242  800
9  142 239  100
10 163 236  500
11 148 232  600
12 128 231 1200
13 156 217 1300
14 129 214 1300
15 146 208  300
16 164 208  900
17 141 206 2100
18 147 193 1000
19 164 193  900
20 129 189 2500
21 155 185 1800
22 139 182  700
;
```

The following PROC OPTMODEL statements read in the data, declare the optimization model, and use the decomposition algorithm to solve it:

```
proc optmodel;
   /* read the node location and demand data */
   set NODES;
   num x {NODES};
   num y {NODES};
   num demand {NODES};
   num capacity = &capacity;
   num num_vehicles = &num_vehicles;
   read data vrpdata into NODES=[node] x y demand;
   set ARCS = {i in NODES, j in NODES: i ne j};
   set VEHICLES = 1..num_vehicles;

   /* define the depot as node 1 */
   num depot = 1;

   /* define the arc cost as the rounded Euclidean distance */
   num cost {<i,j> in ARCS} = round(sqrt((x[i]-x[j])^2 + (y[i]-y[j])^2));

   /* Flow[i,j,k] is the amount of demand carried on arc (i,j) by vehicle k */
   var Flow {ARCS, VEHICLES} >= 0 <= capacity;
   /* UseNode[i,k] = 1, if and only if node i is serviced by vehicle k */
   var UseNode {NODES, VEHICLES} binary;
   /* UseArc[i,j,k] = 1, if and only if arc (i,j) is traversed by vehicle k */
   var UseArc {ARCS, VEHICLES} binary;

   /* minimize the total distance traversed */
   min TotalCost = sum {<i,j> in ARCS, k in VEHICLES} cost[i,j] * UseArc[i,j,k];

   /* each non-depot node must be serviced by at least one vehicle */
   con Assignment {i in NODES diff {depot}}:
      sum {k in VEHICLES} UseNode[i,k] >= 1;

   /* each vehicle must start at the depot node */
   for{k in VEHICLES} fix UseNode[depot,k] = 1;

   /* some vehicle k traverses an arc that leaves node i
      if and only if UseNode[i,k] = 1 */
   con LeaveNode {i in NODES, k in VEHICLES}:
      sum {<(i),j> in ARCS} UseArc[i,j,k] = UseNode[i,k];

   /* some vehicle k traverses an arc that enters node i
      if and only if UseNode[i,k] = 1 */
   con EnterNode {i in NODES, k in VEHICLES}:
      sum {<j,(i)> in ARCS} UseArc[j,i,k] = UseNode[i,k];

   /* the amount of demand supplied by vehicle k to node i must equal demand
      if UseNode[i,k] = 1; otherwise, it must equal 0 */
   con FlowBalance {i in NODES diff {depot}, k in VEHICLES}:
       sum {<j,(i)> in ARCS} Flow[j,i,k] - sum {<(i),j> in ARCS} Flow[i,j,k]
       = demand[i] * UseNode[i,k];

   /* if UseArc[i,j,k] = 1, then the flow on arc (i,j) must be at most capacity
      if UseArc[i,j,k] = 0, then no flow is allowed on arc (i,j) */
```

```
   con VehicleCapacity {<i,j> in ARCS, k in VEHICLES}:
      Flow[i,j,k] <= Flow[i,j,k].ub * UseArc[i,j,k];

   /* decomp by vehicle */
   for {i in NODES, k in VEHICLES} do;
      LeaveNode[i,k].block = k;
      EnterNode[i,k].block = k;
   end;
   for {i in NODES diff {depot}, k in VEHICLES} FlowBalance[i,k].block = k;
   for {<i,j> in ARCS, k in VEHICLES} VehicleCapacity[i,j,k].block = k;

   /* solve using decomp (aggregate formulation) */
   solve with MILP / varsel=ryanfoster decomp=(logfreq=20);
```

The following OPTMODEL statements create node and edge data for the optimal routing:

```
   /* create solution data set */
   str color {k in VEHICLES} =
      ['red' 'green' 'blue' 'black' 'orange' 'gray' 'maroon' 'purple'];
   create data node_data from [i] x y;
   create data edge_data from [i j k]=
      {<i,j> in ARCS, k in VEHICLES: UseArc[i,j,k].sol > 0.5}
      x1=x[i] y1=y[i] x2=x[j] y2=y[j] linecolor=color[k];
quit;
```

The solution summary is displayed in Output 15.8.1.

Output 15.8.1 Solution Summary

The OPTMODEL Procedure

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	TotalCost
Solution Status	Optimal
Objective Value	603
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	7.730705E-12
Bound Infeasibility	9.325873E-15
Integer Infeasibility	9.325873E-15
Best Bound	603
Nodes	1
Iterations	70
Presolve Time	0.17
Solution Time	87.55

The iteration log is displayed in Output 15.8.2.

Output 15.8.2 Log

```
NOTE: There were 22 observations read from the data set WORK.VRPDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 7568 variables (0 free, 8 fixed).
NOTE: The problem has 3872 binary and 0 integer variables.
NOTE: The problem has 4237 linear constraints (3696 LE, 520 EQ, 21 GE, 0 range).
NOTE: The problem has 22528 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 8 variables and 0 constraints.
NOTE: The MILP presolver removed 16 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 7560 variables, 4237 constraints, and 22512 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: All blocks are identical and the master model is set covering.
WARNING: The master model is not a set partitioning and VARSEL=RYANFOSTER. The objective
         function must ensure that there exists at least one optimal solution that fulfills all
         of the master constraints at equality.
NOTE: The Decomposition algorithm is using an aggregate formulation and Ryan-Foster branching.
NOTE: The number of block threads has been reduced to 1 threads.
NOTE: The problem has a decomposable structure with 8 blocks. The largest block covers 12.44%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 7560 (100%) variables and 4216 (99.5%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best       Master          Best       LP        IP  CPU Real
                   Bound    Objective       Integer      Gap      Gap Time Time
NOTE: Starting phase 1.
         1       0.0000      20.0000             . 2.00e+01        .    1    1
        20       0.0000       0.7500             . 7.50e-01        .    6    5
        25       0.0000       0.0000             .    0.00%        .    8    7
NOTE: Starting phase 2.
         .     112.0000     969.0000     969.0000  765.18%  765.18%    8    7
        29     112.0000     884.9730     907.0000  690.15%  709.82%    9    8
        32     112.0000     747.0000     747.0000  566.96%  566.96%   21   14
        39     186.0829     684.8000     747.0000  268.01%  301.43%   40   26
         .     186.0829     656.5000     747.0000  252.80%  301.43%   40   26
        40     186.0829     656.5000     747.0000  252.80%  301.43%   42   27
        41     249.3439     656.5000     747.0000  163.29%  199.59%   44   29
        42     278.7811     656.5000     747.0000  135.49%  167.95%   48   31
        44     297.0744     656.5000     747.0000  120.99%  151.45%   49   33
        45     338.7579     656.5000     747.0000   93.80%  120.51%   53   36
        47     358.8263     656.5000     747.0000   82.96%  108.18%   57   39
        52     455.9563     648.1538     660.0000   42.15%   44.75%   69   48
        56     499.1319     616.4444     660.0000   23.50%   32.23%   77   54
        59     531.4382     610.1538     660.0000   14.81%   24.19%   91   60
```

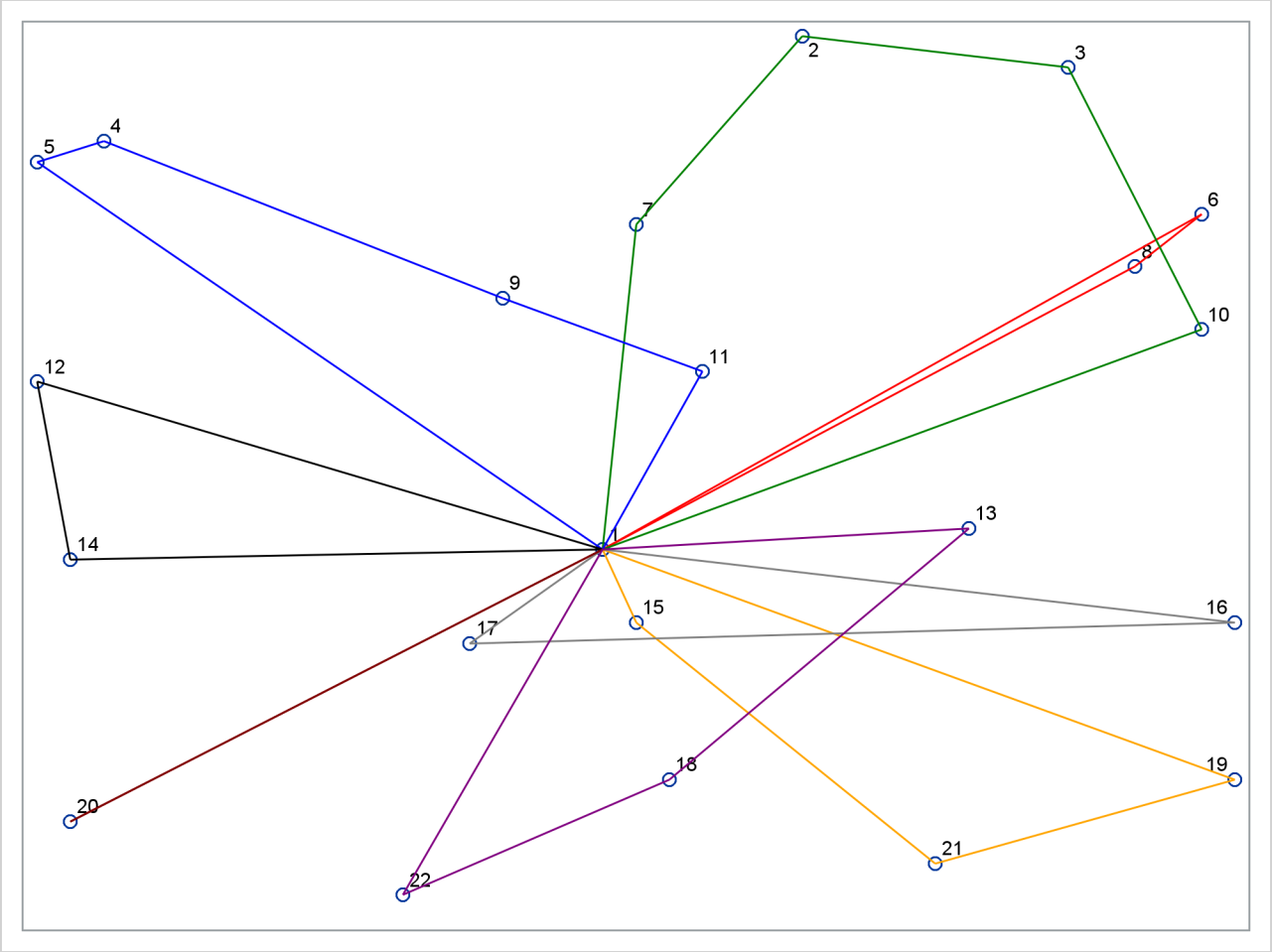
Output 15.8.2 *continued*

```
        60     531.4382     606.6923     660.0000    14.16%    24.19%    95    63
        62     569.3333     604.0000     604.0000     6.09%     6.09%   104    66
        63     572.0000     604.0000     604.0000     5.59%     5.59%   113    70
        67     597.1667     603.8333     604.0000     1.12%     1.14%   140    80
        69     597.1667     603.0000     603.0000     0.98%     0.98%   147    84
        70     603.0000     603.0000     603.0000     0.00%     0.00%   156    87
         Node  Active   Sols          Best          Best       Gap     CPU    Real
                                    Integer         Bound              Time    Time
            0       0      7     603.0000     603.0000     0.00%     156      87
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 87.36 seconds.
NOTE: Optimal.
NOTE: Objective = 603.
NOTE: The data set WORK.NODE_DATA has 22 observations and 3 variables.
NOTE: The data set WORK.EDGE_DATA has 29 observations and 8 variables.
```

The following DATA step and call to PROC SGPLOT generate a plot of the optimal routing. The plot is displayed in Figure 15.8.3.

```
data sganno(drop=i j);
   retain drawspace "datavalue" linethickness 1;
   set edge_data;
   function = 'line';
run;

proc sgplot data=node_data sganno=sganno;
   scatter x=x y=y / datalabel=i;
   xaxis display=none;
   yaxis display=none;
run;
```

Output 15.8.3 Optimal Routing



Example 15.9: ATM Cash Management in Single-Machine Mode

This example describes an optimization model that is used in the management of cash flow for a bank's automated teller machine (ATM) network. The goal of the model is to determine a replenishment schedule for the bank to use in allocating cash inventory at its branches when servicing a preassigned subset of ATMs. Given a history of withdrawals per day for each ATM, the bank can use SAS forecasting tools to predict the expected cash need. The modeling of this prediction depends on various seasonal factors, including the days of the week, weeks of the month, holidays, typical salary disbursement days, location of the ATMs, and other demographic data. The prediction is a parametric mixture of models whose parameters depend on each ATM.

The optimization model performs a polynomial regression that minimizes the error (measured by the L_1 norm) between the predicted and actual withdrawals. The parameter settings in the regression determine the replenishment policy. The amount of cash that is allocated to each day is subject to a budget constraint. In addition, a constraint for each ATM limits the number of days that a *cash-out* (a situation in which the cash flow is less than the predicted withdrawal) can occur. The goal is to determine a policy for cash distribution that balances the predicted inventory levels while satisfying the budget and cash-out constraints. By keeping

too much cash on hand for ATM fulfillment, the bank loses an investment opportunity. Moreover, regulatory agencies in many countries enforce a minimum cash reserve ratio at branch banks; according to regulatory policy, the cash in ATMs or in transit does not contribute toward this threshold.

Mixed Integer Nonlinear Programming Formulation

The most natural formulation for this model is in the form of a mixed integer nonlinear program (MINLP). Let A denote the set of ATMs and D denote the set of days that are used in the training data. The predictive model fit is defined by the following data for each ATM a on each day d: $c_{ad}, c^x_{ad}, c^y_{ad}, c^z_{ad}$, and c^u_{ad}. The model-fitting parameters define the variables (x_a, y_a, u_a) for each ATM that, when applied to the predictive model, estimate the necessary cash flow per day per ATM. In addition, define a surrogate variable f_{ad} for each ATM on each day that defines the cash inventory (replenished from the branch) minus withdrawals. The variable f_{ad} also represents the error in the regression model. Let B_d define the budget per day, K_a define the limit on cash-outs per ATM, and w_{ad} define the historical withdrawals at a particular ATM on a particular day. Then the following MINLP models this problem:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{a \in A} \sum_{d \in D} |f_{ad}| \\
\text{subject to} \quad & c^x_{ad} x_a + c^y_{ad} y_a + \\
& c^z_{ad} x_a y_a + c^u_{ad} u_a + c_{ad} - w_{ad} = f_{ad} && a \in A,\ d \in D && \text{(CashFlowDefCon)} \\
& \sum_{a \in A} (f_{ad} + w_{ad}) \leq B_d && d \in D && \text{(BudgetCon)} \\
& |\{d \in D \mid f_{ad} < 0\}| \leq K_a && a \in A && \text{(CashOutLimitCon)} \\
& x_a, y_a \in [0, 1] && a \in A \\
& u_a \geq 0 && a \in A \\
& f_{ad} \geq -w_{ad} && a \in A,\ d \in D
\end{aligned}
$$

The CashFlowDefCon constraint defines the surrogate variable f_{ad}, which gives the estimated net cash flow. The BudgetCon and CashOutLimitCon constraints ensure that the solution satisfies the budget and cash-out constraints, respectively.

To express this model in a more standard form, you can first use some standard model reformulations to linearize the absolute value and the CashOutLimitCon constraint.

Linearization of Absolute Value

A well-known reformulation for linearizing the absolute value of a variable is to introduce one variable for each side of the absolute value. The following systems are equivalent:

$$
\begin{array}{lll}
\text{minimize} & |y| & \\
\text{subject to} & Ay & \leq \ b
\end{array}
\qquad \text{is equivalent to} \qquad
\begin{array}{lll}
\text{minimize} & y^+ + y^- & \\
\text{subject to} & A(y^+ - y^-) & \leq \ b \\
& y^+, y^- & \geq \ 0
\end{array}
$$

Let f_{ad}^+ and f_{ad}^- represent the positive and negative parts, respectively, of the net cash flow f_{ad}. Then you can rewrite the model, removing the absolute value, as the following:

$$\begin{aligned}
\text{minimize} \quad & \sum_{a \in A} \sum_{d \in D} \left(f_{ad}^+ + f_{ad}^- \right) \\
\text{subject to} \quad & c_{ad}^x x_a + c_{ad}^y y_a + \\
& c_{ad}^z x_a y_a + c_{ad}^u u_a + c_{ad} - w_{ad} = f_{ad}^+ - f_{ad}^- && a \in A,\ d \in D \\
& \sum_{a \in A} \left(f_{ad}^+ - f_{ad}^- + w_{ad} \right) \leq B_d && d \in D \\
& |\{d \in D \mid (f_{ad}^+ - f_{ad}^-) < 0\}| \leq K_a && a \in A \\
& x_a, y_a \in [0, 1] && a \in A \\
& u_a \geq 0 && a \in A \\
& f_{ad}^+ \geq 0 && a \in A,\ d \in D \\
& f_{ad}^- \in [0, w_{ad}] && a \in A,\ d \in D
\end{aligned}$$

Modeling the Cash-Out Constraints

To count the number of times a cash-out occurs, you need to introduce a binary variable to keep track of when this event occurs. Let v_{ad} be an indicator variable that takes the value 1 when the net cash flow is negative. You can model the implication $f_{ad}^- > 0 \Rightarrow v_{ad} = 1$, or its contrapositive $v_{ad} = 0 \Rightarrow f_{ad}^- \leq 0$, by adding the constraint

$$f_{ad}^- \leq w_{ad} v_{ad} \quad a \in A,\ d \in D$$

Now you can model the cash-out constraint by counting the number of days that the net-cash flow is negative for each ATM, as follows:

$$\sum_{d \in D} v_{ad} \leq K_a \quad a \in A$$

The MINLP model can now be written as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{a \in A} \sum_{d \in D} \left(f_{ad}^+ + f_{ad}^- \right) \\
\text{subject to} \quad & c_{ad}^x x_a + c_{ad}^y y_a + \\
& c_{ad}^z x_a y_a + c_{ad}^u u_a + c_{ad} - w_{ad} = f_{ad}^+ - f_{ad}^- && a \in A,\ d \in D \\
& \sum_{a \in A} \left(f_{ad}^+ - f_{ad}^- + w_{ad} \right) \leq B_d && d \in D \\
& f_{ad}^- \leq w_{ad} v_{ad} && a \in A,\ d \in D \\
& \sum_{d \in D} v_{ad} \leq K_a && a \in A \\
& x_a, y_a \in [0, 1] && a \in A \\
& u_a \geq 0 && a \in A \\
& f_{ad}^+ \geq 0 && a \in A,\ d \in D \\
& f_{ad}^- \in [0, w_{ad}] && a \in A,\ d \in D \\
& v_{ad} \in \{0, 1\} && a \in A,\ d \in D
\end{aligned}$$

This MINLP is difficult to solve, in part because the prediction function is not convex. Another approach is to use mixed integer linear programming (MILP) to formulate an approximation of the problem, as described in the next section.

Mixed Integer Linear Programming Approximation

Because the predictive model is a forecast, finding the optimal parameters that are based on nondeterministic data is not of primary importance. Rather, you want to provide as good a solution as possible in a reasonable amount of time. So using MILP to approximate the MINLP is perfectly acceptable. In the original problem you have products of two continuous variables that are both bounded by 0 (lower bound) and 1 (upper bound). This arrangement enables you to create an approximate linear model by using a few standard modeling reformulations.

Discretization of Continuous Variables

The first step is to discretize one of the continuous variables x_a. The goal is to transform the product $x_a y_a$ of a continuous variable and another continuous variable instead to the product of a continuous variable and a binary variable. This transformation enables you to linearize the product form.

You must assume some level of approximation by defining a binary variable (from some discrete set) for each possible setting of the continuous variable For example, if you let $n = 10$, then you allow x to be chosen from the set $\{0.0, 0.1, 0.2, 0.3, ..., 1.0\}$. Let $T = \{0, 1, 2, ..., n\}$ represent the possible steps and $c_t = t/n$. Then you apply the following transformation to variable x_a:

$$\sum_{t \in T} c_t x_{at} = x_a$$

$$\sum_{t \in T} x_{at} = 1$$

$$x_{at} \in \{0, 1\} \quad t \in T$$

The MINLP model can now be approximated as the following:

$$\begin{aligned}
\text{minimize} \quad & \sum_{a\in A}\sum_{d\in D}\left(f_{ad}^{+}+f_{ad}^{-}\right) \\
\text{subject to} \quad & c_{ad}^{x}\sum_{t\in T}c_t x_{at}+c_{ad}^{y}y_a+ \\
& c_{ad}^{z}\sum_{t\in T}c_t x_{at}y_a+c_{ad}^{u}u_a+c_{ad}-w_{ad}=f_{ad}^{+}-f_{ad}^{-} && a\in A,\ d\in D \\
& \sum_{t\in T}x_{at}=1 && a\in A \\
& \sum_{a\in A}\left(f_{ad}^{+}-f_{ad}^{-}+w_{ad}\right)\le B_d && d\in D \\
& f_{ad}^{-}\le w_{ad}v_{ad} && a\in A,\ d\in D \\
& \sum_{d\in D}v_{ad}\le K_a && a\in A \\
& y_a\in[0,1] && a\in A \\
& u_a\ge 0 && a\in A \\
& f_{ad}^{+}\ge 0 && a\in A,\ d\in D \\
& f_{ad}^{-}\in[0,w_{ad}] && a\in A,\ d\in D \\
& v_{ad}\in\{0,1\} && a\in A,\ d\in D \\
& x_{at}\in\{0,1\} && a\in A,\ t\in T
\end{aligned}$$

Linearization of Products

You still need to linearize the product terms $x_{at}y_a$ in the cash flow constraint. Because these terms are products of a bounded continuous variable and a binary variable, you can linearize them by introducing for each product another variable, z_{at}, which serves as a surrogate. In general, you know the following relationship between the original variables and their surrogates:

$$\begin{aligned}
z_t &= x_t y && t\in T \\
\textstyle\sum_{t\in T}x_t &= 1 \\
x_t &\in \{0,1\} && t\in T \\
y &\in [0,1]
\end{aligned}
\qquad \text{is equivalent to} \qquad
\begin{aligned}
z_t &\ge 0 && t\in T \\
z_t &\le x_t && t\in T \\
\textstyle\sum_{t\in T}x_t &= 1 \\
\textstyle\sum_{t\in T}z_t &= y \\
x_t &\in \{0,1\} && t\in T \\
y &\in [0,1]
\end{aligned}$$

Using this relationship to replace each product form, you now can write the problem as an approximate MILP as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{a\in A}\sum_{d\in D}\left(f^{+}_{ad}+f^{-}_{ad}\right) \\
\text{subject to} \quad & c^{x}_{ad}\sum_{t\in T}c_t x_{at}+c^{y}_{ad}y_a+ \\
& c^{z}_{ad}\sum_{t\in T}c_t z_{at}+c^{u}_{ad}u_a+c_{ad}-w_{ad}=f^{+}_{ad}-f^{-}_{ad} && a\in A,\ d\in D \\
& \sum_{t\in T}x_{at}=1 && a\in A \\
& \sum_{a\in A}\left(f^{+}_{ad}-f^{-}_{ad}+w_{ad}\right)\le B_d && d\in D \qquad \text{(BudgetCon)} \\
& f^{-}_{ad}\le w_{ad}v_{ad} && a\in A,\ d\in D \\
& \sum_{d\in D}v_{ad}\le K_a && a\in A \\
& z_{at}\le x_{at} && a\in A,\ t\in T \\
& \sum_{t\in T}z_{at}=y_a && a\in A \\
& z_{at}\ge 0 && a\in A,\ t\in T \\
& y_a\in[0,1] && a\in A \\
& u_a\ge 0 && a\in A \\
& f^{+}_{ad}\ge 0 && a\in A,\ d\in D \\
& f^{-}_{ad}\in[0,w_{ad}] && a\in A,\ d\in D \\
& v_{ad}\in\{0,1\} && a\in A,\ d\in D \\
& x_{at}\in\{0,1\} && a\in A,\ t\in T
\end{aligned}$$

PROC OPTMODEL Code

Because it is difficult to solve the MINLP model directly, the approximate MILP formulation is attractive. Unfortunately, the approximate MILP is much larger than the associated MINLP. Direct methods for solving this MILP do not work well. However, the problem is nicely suited for the decomposition algorithm.

When you examine the structure of the MILP model, you see clearly that the constraints can be easily decomposed by ATM. In fact, the only set of constraints that involve decision variables across ATMs is the BudgetCon constraint. That is, if you relax the budget constraint, you are left with independent blocks of constraints, one for each ATM.

To show how this is done in PROC OPTMODEL, consider the following data sets, which describe an example that tracks 20 ATMs over a period of 100 days. This particular example was submitted to MIPLIB 2010, which is a collection of difficult MILPs in the public domain (Koch et al. 2011).

The first data set, budget_data, provides the cash budget on each particular day:

```
data budget_data;
   input d $ budget;
   datalines;
DATE0      70079
DATE1      66418
DATE10     52656
DATE11     50439
DATE12     58688
DATE13     45002
DATE14     52369
...
;
```

The second data set, cashout_data, provides the limit on the number of cash-outs that are allowed at each ATM:

```
data cashout_data;
   input a $ cashOutLimit;
   datalines;
ATM0            31
ATM1            24
ATM2            41
ATM3            43
ATM4            29
ATM5            24
ATM6            52
ATM7            44
ATM8            35
ATM9            48
ATM10           31
ATM11           47
ATM12           26
ATM13           34
ATM14           29
ATM15           32
ATM16           33
ATM17           32
ATM18           43
ATM19           28
;
```

The final data set, polyfit_data, provides the polynomial fit coefficients for each ATM on each date. It also provides the historical cash withdrawals.

```
data polyfit_data;
   input a $ d $ cx cy cz cu c withdrawal;
   datalines;
ATM0     DATE0       2822     1984    -1984    1045    1373        780
ATM0     DATE1       1337     2530    -2530    1510     174       2351
ATM0     DATE2       2685      -67       67     145    2820       2288
ATM0     DATE3       -595    -3135     3135     581    3319       1357
...
ATM19    DATE96      -734     3392    -3392     162    1648        914
ATM19    DATE97     -1062      969     -969     444    1746       2264
ATM19    DATE98      7676     2308    -2308      59    1388        972
ATM19    DATE99      3062     1308    -1308    1080     654        698
;
```

The following PROC OPTMODEL statements read in the data and define the necessary sets and parameters:

```
proc optmodel;
   set<str> DATES;
   set<str> ATMS;

   /* cash budget per date */
   num budget{DATES};

   /* maximum number of cash-outs allowed at each atm */
   num cashOutLimit{ATMS};

   /* historical withdrawal amount per atm each date */
   num withdrawal{ATMS, DATES};

   /* polynomial fit coefficients for predicted cash flow needed */
   num c {ATMS, DATES};
   num cx{ATMS, DATES};
   num cy{ATMS, DATES};
   num cz{ATMS, DATES};
   num cu{ATMS, DATES};

   /* number of points used in approximation of continuous range */
   num nSteps = 10;
   set STEPS = {0..nSteps};

   read data budget_data into DATES=[d] budget;
   read data cashout_data into ATMS=[a] cashOutLimit;
   read data polyfit_data into [a d] cx cy cz cu c withdrawal;
```

The following statements declare the variables:

```
   var x{ATMS,STEPS}                 binary;
   var v{ATMS,DATES}                 binary;
   var z{ATMS,STEPS}                 >= 0 <= 1;
   var y{ATMS}                       >= 0 <= 1;
   var u{ATMS}                       >= 0;
   var fPlus{ATMS,DATES}             >= 0;
   var fMinus{a in ATMS, d in DATES} >= 0 <= withdrawal[a,d];
```

The following statements declare the objective and the constraints:

```
min CashFlowDiff =
   sum{a in ATMS, d in DATES} (fPlus[a,d] + fMinus[a,d]);

con BudgetCon{d in DATES}:
   sum{a in ATMS} (fPlus[a,d] - fMinus[a,d] + withdrawal[a,d])
       <= budget[d];

con CashFlowDefCon{a in ATMS, d in DATES}:
   cx[a,d] * sum{t in STEPS} (t/nSteps) * x[a,t] +
   cy[a,d] * y[a]                               +
   cz[a,d] * sum{t in STEPS} (t/nSteps) * z[a,t] +
   cu[a,d] * u[a]                               +
   c[a,d] - withdrawal[a,d] = fPlus[a,d] - fMinus[a,d];

con PickOneStepCon{a in ATMS}:
   sum{t in STEPS} x[a,t] = 1;

con CashOutLinkCon{a in ATMS, d in DATES}:
   fMinus[a,d] <= withdrawal[a,d] * v[a,d];

con CashOutLimitCon{a in ATMS}:
   sum{d in DATES} v[a,d] <= cashOutLimit[a];

con Linear1Con{a in ATMS, t in STEPS}:
   z[a,t] <= x[a,t];

con Linear2Con{a in ATMS}:
   sum{t in STEPS} z[a,t] = y[a];
```

The following statements define the block decomposition by ATM. The **`.block`** suffix expects numeric indices, whereas the **`SET<STR> ATMS`** statement declares a set of strings. You can create a mapping from the string identifier to a numeric identifier as follows:

```
/* create numeric block index */
num blockIndex {ATMS};
num index init 0;
for{a in ATMS} do;
   blockIndex[a] = index;
   index = index + 1;
end;
```

Then, each constraint can be added to its associated ATM block as follows:

```
/* define blocks for each ATM */
for{a in ATMS} do;
   PickOneStepCon[a].block  = blockIndex[a];
   CashOutLimitCon[a].block = blockIndex[a];
   Linear2Con[a].block      = blockIndex[a];
   for{d in DATES} do;
      CashFlowDefCon[a,d].block = blockIndex[a];
      CashOutLinkCon[a,d].block = blockIndex[a];
   end;
   for{t in STEPS}
      Linear1Con[a,t].block = blockIndex[a];
end;
```

The budget constraint links all the ATMs, and it remains in the master problem. Finally, the following statements use the decomposition algorithm to solve the problem to within 1% of proven optimality:

```
    /* set the number of threads and get performance details */
    performance details nthreads=4;

    /* solve with the decomposition algorithm */
    solve with milp / relobjgap=0.01 decomp;
quit;
```

The solution summary, performance information, and procedure task timing tables are displayed in Output 15.9.1.

Output 15.9.1 Performance Information, Solution Summary, and Task Timing Tables

The OPTMODEL Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	CashFlowDiff
Solution Status	Optimal within Relative Gap
Objective Value	2463991.7254
Relative Gap	0.0003566183
Absolute Gap	878.39121522
Primal Infeasibility	4.774847E-12
Bound Infeasibility	3.330669E-15
Integer Infeasibility	3.330669E-15
Best Bound	2463113.3342
Nodes	1
Iterations	7
Presolve Time	0.28
Solution Time	68.27

Procedure Task Timing		
Task	**Time (sec.)**	**Time**
Problem Generation	0.04	0.06%
Solver Initialization	0.09	0.13%
Code Generation	0.00	0.00%
Solver	68.28	99.81%
Solver Postprocessing	0.00	0.01%

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 15.9.2.

Output 15.9.2 Log

```
NOTE: There were 100 observations read from the data set WORK.BUDGET_DATA.
NOTE: There were 20 observations read from the data set WORK.CASHOUT_DATA.
NOTE: There were 2000 observations read from the data set WORK.POLYFIT_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6480 variables (0 free, 0 fixed).
NOTE: The problem has 2220 binary and 0 integer variables.
NOTE: The problem has 4380 linear constraints (2340 LE, 2040 EQ, 0 GE, 0 range).
NOTE: The problem has 58878 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 308 variables and 209 constraints.
NOTE: The MILP presolver removed 726 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 6172 variables, 4171 constraints, and 58152 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The problem has a decomposable structure with 20 blocks. The largest block covers 5.131%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 6172 (100%) variables and 4071 (97.6%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best       Master          Best       LP        IP  CPU Real
                   Bound    Objective       Integer      Gap       Gap Time Time
NOTE: Starting phase 1.
         1       0.0000       1.1767             . 1.18e+00         .   63   20
         2       0.0000       0.0000             .    0.00%         .   63   20
NOTE: Starting phase 2.
         3   2.4432e+06   2.6130e+06             .    6.95%         .  101   33
         4   2.4525e+06   2.4831e+06             .    1.25%         .  138   46
         7   2.4631e+06   2.4631e+06             .    0.00%         .  200   67
         .   2.4631e+06   2.4631e+06   2.4640e+06    0.00%     0.04%  200   67
NOTE: The Decomposition algorithm stopped on the integer RELOBJGAP= option.
         Node  Active   Sols         Best          Best       Gap     CPU   Real
                                   Integer         Bound             Time   Time
            0       0      1   2.4640e+06   2.4631e+06     0.04%    200     67
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 67.91 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 2463991.7254.
```

Example 15.10: ATM Cash Management in Distributed Mode

This section illustrates how you can use PROC OPTMODEL and the decomposition algorithm in distributed mode. The problem is the same as the one described in Example 15.9 for managing the cash flow of an ATM network. The only difference between single-machine and distributed mode is that the PERFORMANCE statement specifies the number of threads to use in single-machine mode or the number of threads and nodes to use in distributed mode.

The following statement changes the operating mode to distributed mode:

```
/* set the number of nodes and threads and get performance details */
performance details nodes=5 nthreads=4;
```

The performance information is displayed in Output 15.10.1. When you specify NODES=5 and NTHREADS=4 in the PERFORMANCE statement in distributed mode, each grid node processes up to four threads simultaneously.

Output 15.10.1 Performance Information

Performance Information	
Host Node	<< your grid host >
Execution Mode	Distributed
Number of Compute Nodes	5
Number of Threads per Node	4

The solution summary and procedure task timing tables are displayed in Output 15.10.2.

Output 15.10.2 Solution Summary and Task Timing Tables

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	CashFlowDiff
Solution Status	Optimal within Relative Gap
Objective Value	2477152.1046
Relative Gap	0.0057620003
Absolute Gap	14191.57937
Primal Infeasibility	2.50111E-12
Bound Infeasibility	2.442491E-15
Integer Infeasibility	2.442491E-15
Best Bound	2462960.5253
Nodes	1
Iterations	6
Presolve Time	0.30
Solution Time	16.93

Procedure Task Timing		
Task	**Time (sec.)**	**Time**
Problem Generation	0.04	0.22%
Solver Initialization	0.08	0.48%
Code Generation	0.00	0.01%
Solver	17.13	99.26%
Solver Postprocessing	0.01	0.04%

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 15.10.3.

Output 15.10.3 Log

```
NOTE: There were 100 observations read from the data set WORK.BUDGET_DATA.
NOTE: There were 20 observations read from the data set WORK.CASHOUT_DATA.
NOTE: There were 2000 observations read from the data set WORK.POLYFIT_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6480 variables (0 free, 0 fixed).
NOTE: The problem has 2220 binary and 0 integer variables.
NOTE: The problem has 4380 linear constraints (2340 LE, 2040 EQ, 0 GE, 0 range).
NOTE: The problem has 58878 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 308 variables and 209 constraints.
NOTE: The MILP presolver removed 726 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 6172 variables, 4171 constraints, and 58152 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in the distributed computing environment with 5
      worker nodes.
NOTE: The DECOMP method value USER is applied.
NOTE: The problem has a decomposable structure with 20 blocks. The largest block covers 5.131%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 6172 (100%) variables and 4071 (97.6%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
      Iter          Best       Master          Best       LP        IP Real
                   Bound    Objective       Integer      Gap      Gap Time
NOTE: Starting phase 1.
         1       0.0000       1.1767             . 1.18e+00        .    6
         2       0.0000       0.0000             .    0.00%        .    6
NOTE: Starting phase 2.
         3   2.4432e+06   2.5891e+06             .    5.97%        .    9
         4   2.4550e+06   2.4794e+06             .    1.00%        .   12
         5   2.4630e+06   2.4642e+06             .    0.05%        .   14
NOTE: The Decomposition algorithm stopped on the continuous RELOBJGAP= option.
         .   2.4630e+06   2.4632e+06   2.4772e+06    0.01%    0.58%   14
NOTE: The Decomposition algorithm stopped on the integer RELOBJGAP= option.
         Node  Active   Sols         Best         Best      Gap   Real
                                   Integer        Bound            Time
            0       0      1   2.4772e+06   2.4630e+06    0.58%     14
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 14.08 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 2477152.1046.
NOTE: The data set WORK.PERFINFO has 4 observations and 3 variables.
```

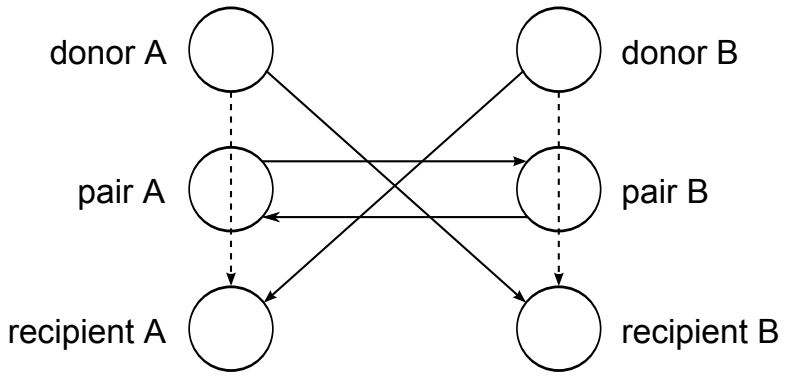
Notice how this iteration log differs from the iteration log from single-machine mode in Example 15.9. In distributed mode, the processing is done on multiple grid machines, as opposed to being done on one client

machine in single-machine mode. In this example, the grid machines and the client machine have different operating systems, and some numerical rounding off leads to different paths in the search space. When you compare two runs on different operating systems (or that use different compilers), this behavior is expected.

Example 15.11: Kidney Donor Exchange and METHOD=SET

This example looks at an application of integer programming to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to donate a kidney. If the donor and recipient are incompatible (because of conflicting blood types, tissue mismatch, and so on), the transplant cannot proceed. Now suppose two donor-recipient pairs, A and B, are in this same situation, but donor A is compatible with recipient B and donor B is compatible with recipient A. Then two transplants can take place in a two-way swap, which is shown graphically in Figure 15.8.

Figure 15.8 Kidney Donor Exchange Two-Way Swap



More generally, an n-way swap that involves n donors and n recipients can be performed (Willingham 2009). To model this problem, define a directed graph as follows. Each node is an incompatible donor-recipient pair. Link (i, j) exists if the donor from node i is compatible with the recipient from node j. Let N define the set of nodes and A define the set of arcs. The link weight, w_{ij}, is a measure of the quality of the match. By introducing dummy links whose weight is 0, you can also include altruistic donors who have no recipients or recipients who have no donors. The idea is to find a maximum-weight node-disjoint union of directed cycles. You want the union to be node-disjoint so that no kidney is donated more than once, and you want cycles so that the donor from node i gives up a kidney if and only if the recipient from node i receives a kidney.

Without any other constraints, the problem could be solved as a linear assignment problem. But doing so would allow arbitrarily long cycles in the solution. Because of practical considerations (such as travel) and to mitigate risk, each cycle must have no more than L links. The kidney exchange problem is to find a maximum-weight node-disjoint union of short directed cycles.

Define an index set $M = \{1, \ldots, |N|/2\}$ of candidate disjoint unions of short cycles (called *matchings*). Let x_{ijm} be a binary variable, which, if set to 1, indicates that arc (i, j) is in a matching m. Let y_{im} be a binary variable that, if set to 1, indicates that node i is covered by matching m. In addition, let s_i be a binary slack variable that, if set to 1, indicates that node i is not covered by any matching.

The kidney donor exchange can be formulated as a MILP as follows:

$$
\begin{array}{lrll}
\text{maximize} & \displaystyle\sum_{(i,j)\in A}\sum_{m\in M} w_{ij}x_{ijm} & & \\
\text{subject to} & \displaystyle\sum_{m\in M} y_{im} + s_i = 1 & i \in N & \text{(Packing)} \\
& \displaystyle\sum_{(i,j)\in A} x_{ijm} = y_{im} & i \in N,\ m \in M & \text{(Donate)} \\
& \displaystyle\sum_{(i,j)\in A} x_{ijm} = y_{jm} & j \in N,\ m \in M & \text{(Receive)} \\
& \displaystyle\sum_{(i,j)\in A} x_{ijm} \le L & m \in M & \text{(Cardinality)} \\
& x_{ijm} \in \{0,1\} & (i,j) \in A,\ m \in M & \\
& y_{im} \in \{0,1\} & i \in N,\ m \in M & \\
& s_i \in \{0,1\} & i \in N &
\end{array}
$$

In this formulation, the Packing constraints ensure that each node is covered by at most one matching. The Donate and Receive constraints enforce the condition that if node i is covered by matching m, then the matching m must use exactly one arc that leaves node i (Donate) and one arc that enters node i (Receive). Conversely, if node i is not covered by matching m, then no arcs that enter or leave node i can be used by matching m. The Cardinality constraints enforce the condition that the number of arcs in matching m must not exceed L.

In this formulation, the matching identifier is arbitrary. Because it is not necessary to cover each incompatible donor-recipient pair (node), the Packing constraints can be modeled by using set partitioning constraints and the slack variable s. Consider a decomposition by matching, in which the Packing constraints form the master problem and all other constraints form identical matching subproblems. As described in the section "Special Case: Identical Blocks and Ryan-Foster Branching" on page 743, this is a situation in which an aggregate formulation and Ryan-Foster branching can greatly improve performance by reducing symmetry.

The following DATA step sets up the problem, first creating a random graph on n nodes with link probability p and Uniform(0,1) weight:

```
/* create random graph on n nodes with arc probability p
   and uniform(0,1) weight */
%let n = 100;
%let p = 0.02;
data ArcData;
   do i = 0 to &n - 1;
      do j = 0 to &n - 1;
         if i eq j then continue;
         else if ranuni(1) < &p then do;
            weight = ranuni(2);
            output;
         end;
      end;
   end;
run;
```

In this case, you can specify METHOD=SET and let the decomposition algorithm automatically detect the set partitioning master constraints (Packing) and each independent matching subproblem. The following PROC OPTMODEL statements read in the data, declare the optimization model, and use the decomposition algorithm to solve it:

```
%let max_length = 10;
proc optmodel;
   set <num,num> ARCS;
   num weight {ARCS};
   read data ArcData into ARCS=[i j] weight;
   print weight;
   set NODES = union {<i,j> in ARCS} {i,j};
   set MATCHINGS = 1..card(NODES)/2;

   /* UseNode[i,m] = 1 if node i is used in matching m, 0 otherwise */
   var UseNode {NODES, MATCHINGS} binary;

   /* UseArc[i,j,m] = 1 if arc (i,j) is used in matching m, 0 otherwise */
   var UseArc {ARCS, MATCHINGS} binary;

   /* maximize total weight of arcs used */
   max TotalWeight
      = sum {<i,j> in ARCS, m in MATCHINGS} weight[i,j] * UseArc[i,j,m];

   /* each node appears in at most one matching */
   /* rewrite as set partitioning (so decomp uses identical blocks)
      sum{} x <= 1 => sum{} x + s = 1, s >= 0 with no associated cost */
   var Slack {NODES} binary;
   con Packing {i in NODES}:
      sum {m in MATCHINGS} UseNode[i,m] + Slack[i] = 1;

   /* at most one recipient for each donor */
   con Donate {i in NODES, m in MATCHINGS}:
      sum {<(i),j> in ARCS} UseArc[i,j,m] = UseNode[i,m];

   /* at most one donor for each recipient */
   con Receive {j in NODES, m in MATCHINGS}:
      sum {<i,(j)> in ARCS} UseArc[i,j,m] = UseNode[j,m];

   /* exclude long matchings */
   con Cardinality {m in MATCHINGS}:
      sum {<i,j> in ARCS} UseArc[i,j,m] <= &max_length;

   /* automatically decompose using METHOD=SET */
   solve with milp / presolver=basic decomp=(method=set);

   /* save solution to a data set */
   create data Solution from
      [m i j]={m in MATCHINGS, <i,j> in ARCS: UseArc[i,j,m].sol > 0.5}
      weight[i,j];
quit;
```

In this case, the PRESOLVER=BASIC option ensures that the model maintains its specified symmetry, enabling the algorithm to use the aggregate formulation and Ryan-Foster branching. The solution summary is displayed in Output 15.11.1.

Output 15.11.1 Solution Summary

The OPTMODEL Procedure

Solution Summary	
Solver	MILP
Algorithm	Decomposition
Objective Function	TotalWeight
Solution Status	Optimal within Relative Gap
Objective Value	26.020287142
Relative Gap	2.1560594E-6
Absolute Gap	0.0000561014
Primal Infeasibility	2.220446E-15
Bound Infeasibility	1.110223E-15
Integer Infeasibility	1.110223E-15
Best Bound	26.020343243
Nodes	19
Iterations	124
Presolve Time	0.02
Solution Time	20.48

The iteration log is displayed in Output 15.11.2.

Output 15.11.2 Log

```
NOTE: There were 194 observations read from the data set WORK.ARCDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 14065 variables (0 free, 0 fixed).
NOTE: The problem has 14065 binary and 0 integer variables.
NOTE: The problem has 9457 linear constraints (48 LE, 9409 EQ, 0 GE, 0 range).
NOTE: The problem has 42001 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value BASIC is applied.
NOTE: The MILP presolver removed 4786 variables and 3298 constraints.
NOTE: The MILP presolver removed 14290 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 9279 variables, 6159 constraints, and 27711 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value SET is applied.
NOTE: All blocks are identical and the master model is set partitioning.
NOTE: The Decomposition algorithm is using an aggregate formulation and Ryan-Foster branching.
NOTE: The number of block threads has been reduced to 1 threads.
NOTE: The problem has a decomposable structure with 48 blocks. The largest block covers 2.062%
      of the constraints in the problem.
NOTE: The decomposition subproblems cover 9216 (99.32%) variables and 6096 (98.98%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
```

Iter	Best Bound	Master Objective	Best Integer	LP Gap	IP Gap	CPU Time	Real Time
.	390.3714	9.2503	9.2503	97.63%	97.63%	0	0
1	388.4993	9.2503	9.2503	97.62%	97.62%	0	0
2	382.6496	10.9240	10.9240	97.15%	97.15%	0	0
3	359.6462	10.9240	10.9240	96.96%	96.96%	0	0
5	359.6462	14.1405	14.1405	96.07%	96.07%	0	0
6	355.2267	21.3961	21.3961	93.98%	93.98%	0	0
7	348.2985	21.3961	21.3961	93.86%	93.86%	0	0
8	330.2734	21.3961	21.3961	93.52%	93.52%	0	0
9	229.6871	22.1269	21.3961	90.37%	90.68%	1	1
.	229.6871	22.1269	21.3961	90.37%	90.68%	1	1
10	229.6871	22.1269	21.3961	90.37%	90.68%	1	1
12	201.2476	23.8995	21.3961	88.12%	89.37%	1	2
16	176.4324	24.5325	21.3961	86.10%	87.87%	2	2
17	159.1914	24.5842	21.3961	84.56%	86.56%	2	2
18	151.9138	24.8491	21.3961	83.64%	85.92%	2	2
19	135.4435	25.0568	21.3961	81.50%	84.20%	2	2
.	135.4435	25.0568	21.9864	81.50%	83.77%	2	2
20	135.4435	25.0568	21.9864	81.50%	83.77%	2	2
22	118.8012	25.2761	22.1565	78.72%	81.35%	2	2
25	109.0404	25.9874	22.1565	76.17%	79.68%	2	2
26	102.9869	26.0423	22.1565	74.71%	78.49%	2	3
27	99.9984	26.0586	22.1565	73.94%	77.84%	2	3

Output 15.11.2 *continued*

```
        28      65.7400      26.2351      22.1565    60.09%    66.30%     3     3
        30      47.2447      26.4827      22.1565    43.95%    53.10%     3     3
        36      41.4607      26.7010      22.1565    35.60%    46.56%     4     4
        39      27.6471      26.7804      22.1565     3.13%    19.86%     4     4
         .      27.6471      26.7804      24.4269     3.13%    11.65%     5     4
        40      26.7804      26.7804      24.4269     0.00%     8.79%     5     5
         .      26.7804      26.7804      24.4269     0.00%     8.79%     5     5
NOTE: Starting branch and bound.
         Node  Active   Sols         Best          Best       Gap     CPU    Real
                                  Integer         Bound             Time    Time
            0       1      9      24.4269      26.7804    8.79%       5       5
            4       6     10      25.9955      26.5360    2.04%      10       9
            8       4     11      26.0203      26.2556    0.90%      16      14
           10       4     11      26.0203      26.2478    0.87%      17      15
           18       0     11      26.0203      26.0203    0.00%      23      20
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 20.40 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 26.020287142.
NOTE: The data set WORK.SOLUTION has 42 observations and 4 variables.
```

The solution is a set of arcs that define a union of short directed cycles (matchings). The following call to PROC OPTNET extracts the corresponding cycles from the list of arcs and outputs them to the data set Cycles:

```
proc optnet
   direction  = directed
   data_links = Solution;
   data_links_var
      from    = i
      to      = j;
   cycle
      mode    = all_cycles
      out     = Cycles;
run;
```

For more information about PROC OPTNET, see *SAS/OR User's Guide: Network Optimization Algorithms*. Alternatively, you can extract the cycles by using the SOLVE WITH NETWORK statement in PROC OPTMODEL (see Chapter 9, "The Network Solver"). The optimal donor exchanges from the output data set Cycles are displayed in Figure 15.9.

Figure 15.9 Optimal Donor Exchanges

cycle=1

order	node
1	5
2	19
3	56
4	12
5	33
6	70
7	63
8	43
9	15
10	5

cycle=2

order	node
1	13
2	74
3	65
4	41
5	59
6	50
7	49
8	98
9	13

cycle=3

order	node
1	16
2	91
3	17
4	57
5	87
6	72
7	64
8	22
9	88
10	16

Figure 15.9 *continued*

cycle=4

order	node
1	8
2	32
3	79
4	71
5	69
6	26
7	9
8	18
9	95
10	35
11	8

cycle=5

order	node
1	52
2	77
3	94
4	81
5	52

cycle=6

order	node
1	24
2	92
3	24

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice-Hall.

Augerat, P., Belenguer, J. M., Benavent, E., Corberán, A., Naddef, D., and Rinaldi, G. (1995). *Computational Results with a Branch and Cut Code for the Capacitated Vehicle Routing Problem*. Technical Report 949-M, Université Joseph Fourier, Grenoble.

Aykanat, C., Pinar, A., and Çatalyürek, Ü. V. (2004). "Permuting Sparse Rectangular Matrices into Block-Diagonal Form." *SIAM Journal on Scientific Computing* 25:1860–1879.

Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., and Vance, P. H. (1998). "Branch-and-Price: Column Generation for Solving Huge Integer Programs." *Operations Research* 46:316–329.

Caprara, A., Furini, F., and Malaguti, E. (2010). *Exact Algorithms for the Temporal Knapsack Problem*. Technical Report OR-10-7, Department of Electronics, Computer Science, and Systems, University of Bologna.

Dantzig, G. B., and Wolfe, P. (1960). "Decomposition Principle for Linear Programs." *Operations Research* 8:101–111. http://www.jstor.org/stable/167547.

Galati, M. V. (2009). *Decomposition in Integer Linear Programming*. Ph.D. diss., Lehigh University.

Gamrath, G. (2010). *Generic Branch-Cut-and-Price*. Diploma thesis, Technische Universität Berlin.

Grcar, J. F. (1990). *Matrix Stretching for Linear Equations*. Technical Report SAND90-8723, Sandia National Laboratories.

Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R. E., Danna, E., Gamrath, G., Gleixner, A. M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D. E., and Wolter, K. (2011). "MIPLIB 2010." *Mathematical Programming Computation* 3:103–163. doi: 10.1007/s12532-011-0025-9. http://mpc.zib.de/index.php/MPC/article/view/56/28.

Ralphs, T. K., and Galati, M. V. (2006). "Decomposition and Dynamic Cut Generation in Integer Linear Programming." *Mathematical Programming* 106:261–285. doi:10.1007/s10107-005-0606-3.

Vanderbeck, F., and Savelsbergh, M. W. P. (2006). "A Generic View of Dantzig-Wolfe Decomposition in Mixed Integer Programming." *Operations Research Letters* 34:296–306. doi:10.1016/j.orl.2005.05.009.

Willingham, V. (2009). "Massive Transplant Effort Pairs 13 Kidneys to 13 Patients." CNN Health. Accessed March 16, 2011. http://www.cnn.com/2009/HEALTH/12/14/kidney.transplant/index.html.

Chapter 16
The OPTMILP Option Tuner

Contents

Overview: The OPTMILP Option Tuner **821**
Getting Started: The OPTMILP Option Tuner **822**
Syntax: The OPTMILP Option Tuner **824**
Functional Summary 824
PERFORMANCE Statement 824
TUNER Statement 825
Details: The OPTMILP Option Tuner **827**
Data Input and Output 827
Default Set of Tuning Options 829
Full Set of Tuning Options 830
Tuner Log 830
ODS Tables 831
Examples: The OPTMILP Option Tuner **833**
Example 16.1: Tuning the Default Set of Options for a Single Problem 833
Example 16.2: Tuning a Defined Set of Options for Multiple Problems 835
Example 16.3: Tuning a Defined Set of Options for Multiple Problems in Distributed Mode 838
References **838**

Overview: The OPTMILP Option Tuner

The OPTMILP procedure provides many solver techniques and algorithms, including branch-and-bound, cutting planes, and heuristics. It also provides control options that you can adjust to improve the performance of these techniques. Although the default values of the control options have been tuned to work well for most instances, you might need to adjust one or more option values for a specific problem. The OPTMILP option tuner is a tool that enables you to explore alternative (and potentially better) option configurations for your optimization problems.

To use the tuner, you specify a single problem or set of problems to be solved and a list of options to be tuned. You can specify initial values for the options to be tuned. The tuner then uses a heuristic local search technique to generate a sequence of configurations. A *configuration* is a set of the specified options to be tuned along with a fixed value for each option. The tuner attempts to locate configurations that enable the OPTMILP procedure to process problems more quickly than the default option values or specified initial values.

The tuner option can be run in either single-machine mode or distributed mode. In single-machine mode, you can specify the number of threads to use on a single computer. In distributed mode, you can specify the number of computer nodes and the number of threads per node to use on a distributed computing environment. The tuner option only supports nondeterministic mode, so the tuning results may vary among different runs.

NOTE: Distributed mode requires SAS High-Performance Optimization software.

Getting Started: The OPTMILP Option Tuner

This example illustrates how to use the OPTMILP option tuner.

The standard set of MILP benchmark cases is called MIPLIB (Bixby et al. 1998; Achterberg, Koch, and Martin 2003) and can be found at `http://miplib.zib.de/`. Suppose you want to solve the problems air04 and air05 from this set. You have stored the SAS data sets *air04* and *air05*, both in MPS format, in library a. Suppose you want to tune the CUTCLIQUE=, CUTGOMORY=, and HEURISTICS options in these two problems.

The following DATA step generates the data set probs, which contains the list of problems to be solved, and the data set optvals, which contains the list of options to be tuned:

```
data probs;
   input name $1-8;
   datalines;
a.air04
a.air05
;

data optvals;
   input option $1-10;
   datalines;
cutclique
cutgomory
heuristics
;
```

The following statements call the OPTMILP procedure and enable the option tuner:

```
proc optmilp maxtime=300;
   tuner maxtime=1200 problems=probs optionvalues=optvals tunerout=out;
   performance nthreads=4;
run;
```

The MAXTIME= option in the PROC OPTMILP statement sets the maximum run time that the procedure can use to solve one problem for each option configuration. The MAXTIME= option in the TUNER statement sets a limit on the total time that the option tuner can use to solve the problems on the list by using the generated sequence of configurations. The PROBLEMS= option specifies the name of the SAS data set that contains the list of problems to be solved. The OPTIONVALUES= option specifies the name of the SAS data set that contains the list of options to be tuned. The TUNEROUT= option specifies the name of the SAS data set that contains detailed results of the tuning process. The NTHREADS= option in the PERFORMANCE

statement specifies the number of threads that the procedure can use to perform calculations. The ODS OUTPUT statement creates an output data set from the TunerResults table.

For more information about the options available in the PROC OPTMILP statement, see the section "PROC OPTMILP Statement" on page 630. For more information about the PERFORMANCE statement, see the section "PERFORMANCE Statement" on page 21.

Figure 16.1 shows a selection of tuning results that include the initial option configuration, the best option configurations, and the worst option configurations.

Figure 16.1 PROC OPTMILP Output

The OPTMILP Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Tuner Information	
Target Solver	MILP
Number of Tuning Options	3
Number of Tuning Instances	2
Tuning Option Set	USER
Performance Goal	GEOMEAN
Tuner Time Limit	1200
Tuner Configurations Limit	2147483647

Tuner Summary	
Actual Tuning Time	1211.95
Initial Run Time (geomean)	27.98
Initial Run Time (sum)	59.44
Best Run Time (geomean)	22.31
Best Run Time (sum)	45.31
Number of Improved Configurations	32
Number of Tested Configurations	75

Tuner Results

	Config 0	Config 1	Config 2	Config 3	Config 4	Config 5	Config 6	Config 7	Config 8	Config 9	Config 10
cutclique	-1	1	1	1	1	1	0	0	0	-1	-1
cutgomory	-1	0	1	0	-1	-1	-1	2	1	2	0
heuristics	-1	0	0	1	0	1	0	1	1	-1	0
Mean of Run Times	27.97	22.31	22.32	22.43	22.55	22.56	49.42	49.61	49.71	15.01	24.35
Sum of Run Times	59.44	45.31	45.37	45.59	45.71	45.87	99.77	100.08	100.31	30.81	50.16
Percentage Successful	100	100	100	100	100	100	100	100	100	50	50

Syntax: The OPTMILP Option Tuner

You can specify the following statements for the option tuner in the OPTMILP procedure:

PROC OPTMILP < *options* > **;**
PERFORMANCE < *performance-options* > **;**
TUNER < *tuner-options* > **;**

Functional Summary

Table 16.1 summarizes the options available for the TUNER statement in the OPTMILP procedure.

Table 16.1 Options for the TUNER Statement

Option	Description
GOAL=	Specifies the goal of the tuning process
LOGFREQ=	Specifies the frequency of printing in the log
LOGLEVEL=	Specifies the detail of tuner progress printed in the log
MAXCONFIGS=	Specifies the maximum number of tuning configurations
MAXTIME=	Specifies the maximum tuning time
OPTIONMODE=	Specifies which set of options to tune
OPTIONVALUES=	Specifies the input data set that contains a list of options to be tuned
PROBLEMS=	Specifies the input data set that contains a list of tuning problems
TUNEROUT=	Specifies the output data set that contains detailed tuning results

The options available for the PROC OPTMILP statement are documented in the section "Functional Summary" on page 629 in Chapter 13, "The OPTMILP Procedure." You must specify the MAXTIME= option in the PROC OPTMILP statement so that the tuner can terminate properly.

PERFORMANCE Statement

PERFORMANCE < *performance-options* > **;**

The PERFORMANCE statement specifies performance options for single-machine mode and distributed mode, passes variables that describe the distributed computing environment, and requests detailed performance results of the OPTMILP procedure.

For single-machine mode, you can use the NTHREADS= option to specify the number of threads to use on a single machine. For distributed mode, you can use the NODES= and NTHREADS= options to specify the numbers of computer nodes and threads per node to use in a distributed computing environment.

When multiple threads are specified, several MILP solvers can run concurrently on a single machine or a computer node. You might consider reducing the value of the NTHREADS= option when the MILP solver returns an out-of-memory status for some tuning problems.

The DETAILS option displays a detailed performance "Timing" table. The OPTMILP option tuner supports only the nondeterministic mode of the PARALLELMODE= option in the PERFORMANCE statement.

The PERFORMANCE statement for single-machine and distributed mode is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

NOTE: Distributed mode requires SAS High-Performance Optimization software.

TUNER Statement

TUNER < *tuner-options* > **;**

You can specify the following options.

GOAL=*number* | *string*
specifies a goal for the option tuner. Table 16.2 describes the valid values of the GOAL= option.

Table 16.2 Values of GOAL= Option

number	*string*	**Description**
0	GEOMEAN	Minimizes the geometric mean of the solution times of the tuning problems
1	SUM	Minimizes the sum of the solution times of the tuning problems

Every attempt to solve a tuning problem that has an option configuration is counted toward the measure that is specified by the GOAL= option.

The default is GEOMEAN. If only one problem is used for option tuning, then GOAL=GEOMEAN and GOAL=SUM are equivalent.

LOGFREQ=*number*
specifies how often tuning information is printed in the log. The value of *number* represents the number of problems solved by the tuner between log updates. The value of *number* can be any nonnegative integer. Specifying LOGFREQ=0 disables log updates. The default is 1.

LOGLEVEL=*number* | *string*
controls the amount of information that the tuner displays in the SAS log. Table 16.3 describes the valid values of the LOGLEVEL= option.

Table 16.3 Values of LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Disables tuner-related messages in the SAS log
1	BASIC	Displays a tuner summary after stopping
2	MODERATE	Prints a tuner summary and a tuning log by using the interval dictated by the LOGFREQ= option

The default is MODERATE.

MAXCONFIGS=*number*

specifies the maximum number of option configurations that the tuner can evaluate in each problem in the PROBLEMS= data set. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default is $2^{31} - 1$. This option is an alternative way for the tuner to control the termination.

MAXTIME=*number*

specifies the maximum time allowed for the tuner to evaluate option configurations in tuning problems. You must specify either this option or the MAXCONFIGS= option so that the tuner can terminate properly.

It is recommended that you specify a value for *number* that is large enough that the tuner can run several different option configurations. This value depends on two quantities: the number of tuning problems and the OPTMILP procedure's average run time for the tuning problems. To prevent the procedure from spending too much time running a single configuration in a single problem, you must limit the time the procedure spends solving each combination of problem and configuration. You can limit the time by specifying the MAXTIME= option in the PROC OPTMILP statement. If you prefer not to stop the option tuner as a result of elapsed time, you can specify the MAXCONFIGS= option.

OPTIONMODE=*number* | *string*

specifies which set of options to tune. Table 16.4 describes the valid values of the OPTIONMODE= option.

Table 16.4 Values of OPTIONMODE= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses an option set that is determined by the tuner
0	NONE	Solves the problems that are specified by the PROBLEMS= option without tuning any OPTMILP options
1	USER	Uses the option set that is specified by the OPTIONVALUES= option
2	FULL	Uses the full set of solver options that are available for tuning

The tuner interprets the OPTIONMODE= option in accordance with the following logic:

1. If you specify neither the OPTIONMODE= nor the OPTIONVALUES= option, the tuner runs with OPTIONMODE=AUTOMATIC.
2. If you specify the OPTIONVALUES= option, you are not required to specify the OPTIONMODE= option, but you can specify OPTIONMODE=USER. Specifying any other value for the OPTIONMODE= option causes the tuner to terminate with an error.
3. If you do not specify the OPTIONVALUES= option, you can specify either OPTIONMODE=FULL or OPTIONMODE=AUTOMATIC. Specifying OPTIONMODE=USER causes the tuner to terminate with an error.

When OPTIONMODE=NONE, the options PRIMALOUT= and DUALOUT= of PROC OPTMILP can be used to output primal and dual solutions of all the problems that are listed in the PROBLEMS=

data set. All primal or dual solutions are appended to the PRIMALOUT= or DUAL= data set. An additional variable _PROBLEM_ is created for each data set to store problem names.

OPTIONVALUES=*SAS-data-set*

OPTVALS=*SAS-data-set*

specifies an input data set that contains a list of options to be tuned and ranges of values over which each option should be tuned. You can specify an initial tuning value for each option in the list. If you do not specify a range for a tuning option, the tuner uses the default range of that option. If you do not specify an initial value for a tuning option, the tuner uses the default value of that option. If the option's default value is not in the specified tuning range, the tuner uses the first (smallest) value in the tuning range. If the data set that is specified by the OPTIONVALUES= option is not found, a default set of options is used. For more information, see the section "Variables in the OPTIONVALUES= Data Set" on page 828 and the section "Default Set of Tuning Options" on page 829.

NOTE: An option value that you specify in the PROC OPTMILP statement is applied to all tuning problems unless you specify that option in the OPTIONVALUES= data set. In that case, the value that you specify in the PROC OPTMILP statement is ignored.

PROBLEMS=*SAS-data-set*

PROBS=*SAS-data-set*

specifies the input data set that contains a list of MILP problems to be used for option tuning. This list includes the name of each problem, its library location, and (optionally) its objective sense. For more information, see the section "Variables in the PROBLEMS= Data Set" on page 828. The tuning problems should be stored in MPS-format SAS data sets. To perform option tuning on a single problem, you can omit the PROBLEMS= option in the TUNER statement and specify the DATA= option in the PROC OPTMILP statement. For more information about this option, see the section "Data Set Options" on page 630.

TUNEROUT=*SAS-data-set*

TOUT=*SAS-data-set*

specifies the output data set to contain detailed results for each tuning problem over all the option configurations that are evaluated. This data set helps you rank the performance with your own rules, especially when some problems are not optimal but the integer solutions are acceptable for some configurations. For more information, see the section "Variables in the TUNEROUT= Data Set" on page 828.

Details: The OPTMILP Option Tuner

Data Input and Output

This subsection describes the input data sets that are specified by the PROBLEMS= and OPTIONVALUES= options and the output data set that is specified by the TUNEROUT= option.

When you specify OPTIONMODE=NONE, you can specify the PRIMALOUT= and DUALOUT= options in the PROC OPTMILP statement to output primal and dual solutions of all the problems listed in the PROBLEMS= data set. All primal or dual solutions are appended to the PRIMALOUT= or DUALOUT=

data set, respectively. An additional variable, _PROBLEM_, is created for each data set; this variable stores problem names.

Variables in the PROBLEMS= Data Set

The PROBLEMS= data set contains the following variables:

NAME
specifies a list of names of MPS-format data sets. Each data set contains a MILP problem to be used in option tuning. The format of each name must be *libref.filename*. If no libref is specified, the tuner searches for the file in WORK.

OBJSENSE
specifies whether the objective sense for a tuning problem is MIN or MAX. The values of this variable provide or overwrite the objective sense for the corresponding SAS data set. This variable is optional.

Variables in the OPTIONVALUES= Data Set

The OPTIONVALUES= data set contains the following variables:

INITIAL
specifies an initial value for each option to be tuned. This variable is optional. If this variable is missing, the tuner uses the default value of the option as the initial value. If the default value of the option is not in the list specified by the VALUES variable, the tuner uses the first entry in the VALUES list for that option.

OPTION
specifies a list of control options to be tuned by solving the problems specified in the PROBLEMS= data set.

VALUES
specifies a comma-delimited list of values for each control option that the tuner can use to generate configurations. If you do not specify a list of values for an option, the tuner uses all valid values of that option. This variable is optional. If you do not provide discrete values, such as (1, 2, 3, 100), for the options that have unlimited number of possible values, then those options are not tuned. An exception is the STRONGITER= option, where the default tuning values are (–1, 100, 10000, 2147483647).

Variables in the TUNEROUT= Data Set

The TUNEROUT= data set contains the following variables:

OBJSENSE
specifies the objective sense used for each tuning problem. The value is either MIN or MAX.

PROBLEM
specifies a list of data set names. Each named data set contains one of the problems that are used by the tuner.

RANK

specifies the rank of each option configuration, based on the criteria specified by the GOAL= option. The row that has a rank value of 0 contains the solution information of either the initial values that you provide or the solver default values for the options. When a solution status is not optimal, feasible, or bounded, a penalty is applied to the solution time in the ranking. If you want to put other factors (such as the relative gap) into the ranking, then you can define your own rules by using the solution information in this data set.

The tuner's ranking is based on solver's running time, which has nondeterministic nature, so the tuning results may vary among different runs.

Option Configurations

name each variable for a tuning option and contain the option value that is used for the current option configuration.

Solution Information

specifies solution information for each option configuration that the tuner evaluates. This information includes the status, solution status, objective value, relative gap, absolute gap, nodes, and solution time. For more information about these terms, see the section "Macro Variable _OROPTMILP_" on page 654. When the tuner's time limit is reached, any unfinished runs will have the solution status TUNER_TIME_LIM.

Default Set of Tuning Options

Table 16.5 lists the options and values that the tuner uses when OPTIONMODE=AUTOMATIC.

Table 16.5 Default Set of Tuning Options

Option	Values
CONFLICTSEARCH=	–1, 0
CUTGOMORY=	–1, 0
CUTMILIFTED=	–1, 0
CUTSTRATEGY=	–1, 0, 1, 2
CUTZEROHALF=	–1, 0
HEURISTICS=	–1, 0, 1, 2, 3
NODESEL=	–1, 0, 1, 2
PRESOLVER=	–1, 0, 1, 2, 3
PROBE=	–1, 0
RESTARTS=	–1, 0, 1, 2, 3
SYMMETRY=	–1, 0, 1, 2, 3
VARSEL=	–1, 3

For more information about these options, see the section "Functional Summary" on page 629 in Chapter 13, "The OPTMILP Procedure."

Full Set of Tuning Options

When OPTIONMODE=FULL, the tuner tunes the set of options listed in Table 16.6 over an automatically determined range.

Table 16.6 Full Set of Tuning Options

ALLCUTS=	CUTSFACTOR=
CONFLICTSEARCH=	CUTSTRATEGY=
CUTCLIQUE=	CUTZEROHALF=
CUTFLOWCOVER=	HEURISTICS=
CUTFLOWPATH=	NODESEL=
CUTGOMORY=	PRESOLVER=
CUTGUB=	PROBE=
CUTIMPLIED=	RESTARTS=
CUTKNAPSACK=	STRONGITER=
CUTLAP=	SYMMETRY=
CUTMILIFTED=	VARSEL=
CUTMIR=	

For more information about these options, see the section "Functional Summary" on page 629 in Chapter 13, "The OPTMILP Procedure." You can also tune other performance related OPTMILP options that are not listed here by using the OPTIONVALUES= option.

Tuner Log

The following information about the option tuner is printed in the tuner log:

SolveCalls	indicates the number of problems that the tuner has completed.
Configurations	indicates the number of configurations that the tuner has completed.
BestTime	indicates the geometric mean or sum of the solve times (over all tuning problems) of the current best option configuration. When one of the solves comes from an unsuccessful run, an asterisk (*) is placed next to the time.
Time	indicates the time (in seconds) that is used by the tuner.

The LOGFREQ= and LOGLEVEL= options can be used to control the amount of information printed in the tuner log. Figure 16.2 shows a sample tuner log.

Figure 16.2 Sample Option Tuner Log

```
NOTE: The OPTMILP procedure is executing in single-machine mode.
NOTE: The Option Tuning algorithm (the Tuner) is enabled.
NOTE: The non-deterministic parallel mode is enabled.
NOTE: The Tuner is using up to 4 threads.
      SolveCalls  Configurations    BestTime        Time
              10               5       22.44       85.56
              20              10       22.44      169.82
              30              15       22.44      254.70
              40              19       22.44      361.15
              50              24       22.44      421.78
              60              29       22.31      504.77
              70              34       22.31      587.31
              80              39       22.31      656.76
              90              45       22.31      731.63
             100              49       22.31      822.75
             110              54       22.31      898.13
             120              59       22.31      986.35
             130              64       22.31     1060.62
             140              70       22.31     1141.70
             150              75       22.31     1208.33
NOTE: The tuning time is 1211.95 seconds.
NOTE: The data set WORK.OUT has 150 observations and 13 variables.
```

ODS Tables

The tuner creates several Output Delivery System (ODS) tables by default unless you specify a value other than 1 for the PRINTLEVEL= option in the PROC OPTMILP statement. The names of these tables are listed in Table 16.7. The TunerInfo and TunerSummary tables contain the tuner's input summary and results summary, respectively. The TunerResults table contains the option values, geometric mean and summation of the solution times, success rate for the initial option configuration, and a selection of best option configurations and worst option configurations.They are sorted according to the success rate and performance measure specified by the GOAL= option. The *Config 0* column contains the solution information of either the initial values that you provided or the solver default values for the options.

The "Performance Information" table is produced by default. It displays information about the execution mode. For single-machine mode, the table displays the number of threads used. For distributed mode, the table displays the grid mode (symmetric or asymmetric), the number of compute nodes, and the number of threads per node.

If you specify the DETAILS option in the PERFORMANCE statement, the procedure also produces a "Timing" table in which the accumulated elapsed times (absolute and relative) for the main tasks of the procedure are displayed.

You can create output data sets from these tables by specifying the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: User's Guide*.

Table 16.7 ODS Tables Produced by the OPTMILP Option Tuner

ODS Table Name	Description
PerformanceInfo	Performance information
Timing	Timing report
TunerInfo	Summary of option tuning input
TunerResults	Option tuning results
TunerSummary	Summary of option tuning results

Figure 16.3 shows an example PerformanceInfo table for single-machine mode.

Figure 16.3 Example Tuner Output: PerformanceInfo

The OPTMILP Procedure

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Figure 16.4 shows an example Timing table.

Figure 16.4 Example Tuner Output: Timing

Procedure Task Timing		
Task	**Time (sec.)**	**Time**
Data Loading	0.55	0.01%
Data Transfer	0.00	0.00%
Tuner	10.83	0.22%
Solver	4829.00	99.73%
Idle	1.52	0.03%

Figure 16.5 shows an example TunerInfo table.

Figure 16.5 Example Tuner Output: TunerInfo

Tuner Information	
Target Solver	MILP
Number of Tuning Options	3
Number of Tuning Instances	2
Tuning Option Set	USER
Performance Goal	GEOMEAN
Tuner Time Limit	1200
Tuner Configurations Limit	2147483647

Figure 16.6 shows an example TunerResults table.

Figure 16.6 Example Tuner Output: TunerResults

Tuner Results											
	Config 0	**Config 1**	**Config 2**	**Config 3**	**Config 4**	**Config 5**	**Config 6**	**Config 7**	**Config 8**	**Config 9**	**Config 10**
cutclique	-1	1	1	1	1	1	0	0	0	-1	-1
cutgomory	-1	0	1	0	-1	-1	-1	2	1	2	0
heuristics	-1	0	0	1	0	1	0	1	1	-1	0
Mean of Run Times	27.97	22.31	22.32	22.43	22.55	22.56	49.42	49.61	49.71	15.01	24.35
Sum of Run Times	59.44	45.31	45.37	45.59	45.71	45.87	99.77	100.08	100.31	30.81	50.16
Percentage Successful	100	100	100	100	100	100	100	100	100	50	50

Figure 16.7 shows an example TunerSummary table.

Figure 16.7 Example Tuner Output: TunerSummary

Tuner Summary	
Actual Tuning Time	1211.95
Initial Run Time (geomean)	27.98
Initial Run Time (sum)	59.44
Best Run Time (geomean)	22.31
Best Run Time (sum)	45.31
Number of Improved Configurations	32
Number of Tested Configurations	75

Examples: The OPTMILP Option Tuner

Example 16.1: Tuning the Default Set of Options for a Single Problem

This example demonstrates how to tune the default set of tuning options for a single problem. The problem is the air05 problem from the MIPLIB 2003 problem set, which is introduced in the section "Getting Started: The OPTMILP Option Tuner" on page 822. The SAS data set that defines the problem (in MPS format) is named *air05*.

Because you are using only one problem to perform option tuning, you do not need to create a PROBLEMS= data set. Because you are tuning the default set of options, you do not need to create an OPTIONVALUES= data set. The following statements call the OPTMILP option tuner and determine the stopping criterion by specifying the MAXCONFIGS= option instead of the MAXTIME= option:

```
proc optmilp data=a.air05 maxtime=300;
   tuner maxconfigs=20 printfreq=2 tunerout=out;
   performance nthreads=4;
run;
```

```
title "Tuner Output";
proc print data=out(obs=10);
run;
```

The output data set is shown in Figure 16.1.1.

Output 16.1.1 Single Problem with Default Tuning Options: Output

Tuner Output

Obs	RANK	PROBLEM	OBJSENSE	PRESOLVER	PROBE	RESTARTS	CONFLICTSEARCH	NODESEL	VARSEL
1	0	AIR05	MIN	-1	-1	-1	-1	-1	-1
2	1	AIR05	MIN	0	0	3	-1	1	-1
3	2	AIR05	MIN	1	0	0	0	0	-1
4	3	AIR05	MIN	3	0	3	0	0	-1
5	4	AIR05	MIN	3	0	3	0	0	-1
6	5	AIR05	MIN	3	0	3	-1	0	-1
7	6	AIR05	MIN	3	0	3	0	0	-1
8	7	AIR05	MIN	3	0	3	0	0	-1
9	8	AIR05	MIN	3	0	3	0	0	-1
10	9	AIR05	MIN	3	0	3	0	0	-1

Obs	HEURISTICS	CUTSTRATEGY	CUTGOMORY	CUTMILIFTED	CUTZEROHALF	SYMMETRY	STATUS
1	-1	-1	-1	-1	-1	-1	OK
2	0	-1	-1	0	0	2	OK
3	2	2	-1	0	0	-1	OK
4	-1	-1	-1	-1	-1	0	OK
5	-1	-1	-1	-1	-1	3	OK
6	-1	-1	-1	-1	-1	3	OK
7	-1	-1	0	-1	-1	3	OK
8	-1	-1	-1	-1	-1	-1	OK
9	2	-1	-1	-1	-1	3	OK
10	-1	1	-1	-1	-1	3	OK

Obs	SOLUTION_STATUS	OBJECTIVE	RELATIVE_GAP	ABSOLUTE_GAP	NODES	SOLUTION_TIME
1	OPTIMAL_RGAP	26374	.000082710	2.18123	1194	39.06
2	OPTIMAL	26374	0	0.00000	297	20.56
3	OPTIMAL_RGAP	26374	.000050350	1.32787	245	29.35
4	OPTIMAL_RGAP	26374	.000088107	2.32353	259	31.04
5	OPTIMAL_RGAP	26374	.000088107	2.32353	259	31.49
6	OPTIMAL_RGAP	26374	.000088107	2.32353	259	31.55
7	OPTIMAL_RGAP	26374	.000088107	2.32353	259	31.57
8	OPTIMAL_RGAP	26374	.000088107	2.32353	259	31.71
9	OPTIMAL_RGAP	26374	.000088107	2.32353	259	31.75
10	OPTIMAL_RGAP	26374	.000089946	2.37202	281	31.77

Example 16.2: Tuning a Defined Set of Options for Multiple Problems

This example demonstrates how to specify a set of tuning options and tune them for multiple problems.

The following DATA step creates a PROBLEMS= data set named probs that contains the list of tuning problems. This data set is the same as in the section "Getting Started: The OPTMILP Option Tuner" on page 822.

```
data probs;
   input name $1-8;
   datalines;
a.air04
a.air05
;
```

The following DATA step creates an OPTIONVALUES= data set named optvals that is different from the default set, which is described in the section "Default Set of Tuning Options" on page 829:

```
data optvals;
   input option $1-10 values $12-28 initial $30-32;
   datalines;
cutclique   -1, 0, 2          -1
cutgomory                      1
heuristics
;
```

The optvals data set contains a nondefault list of tuning values for the CUTCLIQUE= option in addition to initial values for the CUTCLIQUE= and CUTGOMORY= options. The options for which sets of tuning values are not specified (in this case, the CUTGOMORY= and HEURISTICS= options) are tuned for all available values if the number of values is finite. The options for which initial values are not specified (in this case, the HEURISTICS= option) are tuned by using the default initial value.

The following statements call the OPTMILP option tuner and then print the ODS table TunerResults and the TUNEROUT= data set:

```
proc optmilp maxtime=300;
   tuner problems=probs  optionvalues=optvals  optionmode=user
         maxtime=1200 tunerout=out;
   performance nthreads=4;
run;

title "Tuner Output";
proc print data=out(obs=20);
run;
```

The output is shown in Figure 16.2.1.

Output 16.2.1 Multiple Problems with Specified Tuning Options: Output

Tuner Output

The OPTMILP Procedure

	Tuner Results										
	Config 0	**Config 1**	**Config 2**	**Config 3**	**Config 4**	**Config 5**	**Config 6**	**Config 7**	**Config 8**	**Config 9**	**Config 10**
cutclique	-1	-1	-1	-1	-1	-1	0	0	0	0	0
cutgomory	1	2	1	1	0	0	0	2	-1	1	2
heuristics	-1	0	1	0	0	1	1	1	1	1	0
Mean of Run Times	28.91	25.66	25.76	25.9	25.91	25.94	49.2	49.27	49.59	49.65	49.72
Sum of Run Times	61.52	54.46	54.11	54.14	54.29	54.33	99.3	99.53	100.06	100.15	100.26
Percentage Successful	100	100	100	100	100	100	100	100	100	100	100

Output 16.2.1 *continued*

Tuner Output

Obs	RANK	PROBLEM	OBJSENSE	cutclique	cutgomory	heuristics	STATUS	SOLUTION_STATUS
1	0	air04	MIN	-1	1	-1	OK	OPTIMAL_RGAP
2	0	air05	MIN	-1	1	-1	OK	OPTIMAL_RGAP
3	1	air04	MIN	-1	2	0	OK	OPTIMAL_RGAP
4	1	air05	MIN	-1	2	0	OK	OPTIMAL
5	2	air04	MIN	-1	1	1	OK	OPTIMAL_RGAP
6	2	air05	MIN	-1	1	1	OK	OPTIMAL
7	3	air04	MIN	-1	1	0	OK	OPTIMAL_RGAP
8	3	air05	MIN	-1	1	0	OK	OPTIMAL
9	4	air04	MIN	-1	0	0	OK	OPTIMAL_RGAP
10	4	air05	MIN	-1	0	0	OK	OPTIMAL
11	5	air04	MIN	-1	0	1	OK	OPTIMAL_RGAP
12	5	air05	MIN	-1	0	1	OK	OPTIMAL
13	6	air04	MIN	-1	2	1	OK	OPTIMAL_RGAP
14	6	air05	MIN	-1	2	1	OK	OPTIMAL
15	7	air04	MIN	-1	-1	1	OK	OPTIMAL_RGAP
16	7	air05	MIN	-1	-1	1	OK	OPTIMAL
17	8	air04	MIN	-1	-1	0	OK	OPTIMAL_RGAP
18	8	air05	MIN	-1	-1	0	OK	OPTIMAL
19	9	air04	MIN	2	2	0	OK	OPTIMAL_RGAP
20	9	air05	MIN	2	2	0	OK	OPTIMAL_RGAP

Obs	OBJECTIVE	RELATIVE_GAP	ABSOLUTE_GAP	NODES	SOLUTION_TIME
1	56137.00	.000089076	5.00000	125	20.27
2	26374.00	.000082710	2.18123	1194	41.25
3	56137.00	.000089076	5.00000	162	18.11
4	26374.00	0	0.00000	858	36.35
5	56137.00	.000089076	5.00000	193	18.79
6	26374.00	0	0.00000	859	35.32
7	56137.00	.000089076	5.00000	193	19.21
8	26374.00	0	0.00000	859	34.92
9	56137.00	.000089076	5.00000	193	19.05
10	26374.00	0	0.00000	859	35.24
11	56137.00	.000089076	5.00000	193	19.09
12	26374.00	0	0.00000	859	35.24
13	56137.00	.000089076	5.00000	162	18.11
14	26374.00	0	0.00000	858	37.23
15	56137.00	.000089076	5.00000	193	19.19
16	26374.00	0	0.00000	859	35.78
17	56137.00	.000089076	5.00000	193	20.11
18	26374.00	0	0.00000	859	36.60
19	56137.00	.000089076	5.00000	162	17.90
20	26374.00	.000062540	1.64932	1170	41.53

Example 16.3: Tuning a Defined Set of Options for Multiple Problems in Distributed Mode

This example demonstrates how to run the tuner in distributed mode. The example is similar to Example 16.2. The only difference between single-machine and distributed mode is that the PERFORMANCE statement specifies the number of threads and nodes to be used. The following statement changes the operating mode to distributed:

```
/* set the numbers of nodes and threads and get performance details */
performance nodes=5 nthreads=4 details;
```

The performance information and procedure task timing tables are displayed in Figure 16.3.1. The NODES=5 and NTHREADS=4 options in the PERFORMANCE statement cause the tuner to run in distributed mode, where each computer node processes up to four threads simultaneously.

Output 16.3.1 Performance Information in Distributed Mode: Output

Tuner Output

Performance Information	
Host Node	<< your grid host >>
Execution Mode	Distributed
Number of Compute Nodes	5
Number of Threads per Node	4

Tuner Output

The OPTMILP Procedure

Procedure Task Timing		
Task	**Time (sec.)**	**Time**
Data Loading	1.42	0.05%
Data Transfer	0.18	0.01%
Tuner	0.74	0.02%
Solver	3065.59	99.84%
Idle	2.46	0.08%

References

Achterberg, T., Koch, T., and Martin, A. (2003). "MIPLIB 2003." http://miplib.zib.de/.

Bixby, R. E., Ceria, S., McZeal, C. M., and Savelsbergh, M. W. P. (1998). "An Updated Mixed Integer Programming Library: MIPLIB 3.0." *Optima* 58:12–15.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stuetzle, T. (2009). "ParamILS: An Automatic Algorithm Configuration Framework." *Journal of Artificial Intelligence Research* 36:267–306.

Chapter 17
The MPS-Format SAS Data Set

Contents

Overview: MPS-Format SAS Data Set **839**
Observations 840
Order of Sections 840
Sections Format: MPS-Format SAS Data Set **841**
NAME Section 841
ROWS Section 841
COLUMNS Section 842
RHS Section (Optional) 843
RANGES Section (Optional) 844
BOUNDS Section (Optional) 845
BRANCH Section (Optional) 847
QSECTION Section (Optional) 847
ENDATA Section 848
Details: MPS-Format SAS Data Set **848**
Converting an MPS/QPS-Format File: %MPS2SASD 848
Length of Variables 849
Examples: MPS-Format SAS Data Set **850**
Example 17.1: MPS-Format Data Set for a Product Mix Problem 850
Example 17.2: Fixed-MPS-Format File 851
Example 17.3: Free-MPS-Format File 851
Example 17.4: Using the %MPS2SASD Macro 852
References **854**

Overview: MPS-Format SAS Data Set

The MPS file format is a format commonly used in industry for describing linear programming (LP) and integer programming (IP) problems (Murtagh 1981; IBM 1988). It can be extended to the QPS format (Maros and Mészáros 1999), which describes quadratic programming (QP) problems. MPS-format and QPS-format files are in text format and have specific conventions for the order in which the different pieces of the mathematical model are specified. The MPS-format SAS data set corresponds closely to the format of an MPS-format or QPS-format file and is used to describe linear programming, mixed integer programming, and quadratic programming problems for SAS/OR.

Observations

An MPS-format data set contains six variables: field1, field2, field3, field4, field5, and field6. The variables field4 and field6 are numeric type; the others are character type. Among the character variables, only the value of field1 is case-insensitive and leading blanks are ignored. Values of field2, field3, and field5 are case-sensitive and leading blanks are NOT ignored. Not all variables are used in a particular observation.

Observations in an MPS-format SAS data set are grouped into sections. Each section starts with an *indicator record*, followed by associated *data records*. Indicator records specify the names of sections and the format of the following data records. Data records contain the actual data values for a section.

Order of Sections

Sections of an MPS-format SAS data set must be specified in a **fixed** order.

Sections of linear programming problems are listed in the following order:

- NAME
- ROWS
- COLUMNS
- RHS (optional)
- RANGES (optional)
- BOUNDS (optional)
- ENDATA

Sections of quadratic programming problems are listed in the following order:

- NAME
- ROWS
- COLUMNS
- RHS (optional)
- RANGES (optional)
- BOUNDS (optional)
- QSECTION (optional)
- ENDATA

Sections of mixed integer programming problems are listed in the following order:

- NAME
- ROWS
- COLUMNS
- RHS (optional)
- RANGES (optional)
- BOUNDS (optional)
- BRANCH (optional)
- ENDATA

Sections Format: MPS-Format SAS Data Set

The following subsections describe the format of the records for each section of the MPS-format data set. Note that each section contains two types of records: an indicator record and multiple data records. The following subsections of this documentation describe the two different types of records for each section of the MPS data set.

NAME Section

The NAME section contains only a single record identifying the name of the problem.

Field1	Field2	Field3	Field4	Field5	Field6
NAME	Blank	Input model name	.	Blank	.

ROWS Section

The ROWS section contains the name and type of the rows (linear constraints or objectives). The type of each row is specified by the indicator code in field1 as follows:

- **MIN**: minimization objective
- **MAX**: maximization objective
- **N**: objective
- **G**: $\geq$ constraint
- **L**: $\leq$ constraint
- **E**: = constraint

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
ROWS	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Indicator code	Row name	Blank	.	Blank	.

Notes:

1. At least one objective row should be specified in the ROWS section. It is possible to specify multiple objective rows. However, among all the data records indicating the objective, only the first one is regarded as the objective row, while the rest are ignored. If a type-N row is taken as the objective row, minimization is assumed.
2. Duplicate entries of field2 in the ROWS section are not allowed. In other words, row name is unique. The variable field2 in the ROWS section cannot take a missing value.

COLUMNS Section

The COLUMNS section defines the column (i.e., variable or decision variable) names of the problem. It also specifies the coefficients of the columns for each row.

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
COLUMNS	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Blank	Column name (e.g., col)	Row name (e.g., rowi)	Matrix element in row rowi, column col	Row name (e.g., rowj)	Matrix element in row rowj, column col

Notes:

1. All elements belonging to one column must be grouped together.
2. A missing coefficient value is ignored. A data record with missing values in both field4 and field6 is ignored.
3. Duplicate entries in each pair of column and row are not allowed.
4. When a sequence of data records have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. The value in field2 of the first data record in the section cannot be missing.

Mixed Integer Programs

Mixed integer programming (MIP) problems require you to specify which variables are constrained to be integers. Integer variables can be specified in the COLUMNS section with the use of special marker records in the following form:

Field1	Field2	Field3	Field4	Field5	Field6
Blank	Marker name	'MARKER' (including the quotation marks)	.	'INTORG' or 'INTEND' (including the quotation marks)	.

A marker record with field5 that contains the value 'INTORG' indicates the start of integer variables. In the marker record that indicates the end of integer variables, field5 must be 'INTEND'. An alternative way to specify integer variables without using the marker records is described in the section "BOUNDS Section (Optional)" on page 845.

Notes:

1. INTORG and INTEND markers must appear in pairs in the COLUMNS section. The marker pairs can appear any number of times.
2. The marker name in field2 should be different from the preceding and following column names.
3. All variables between the INTORG and INTEND markers are assumed to be binary unless you specify a different lower bound and/or upper bound in the BOUNDS section.

RHS Section (Optional)

The RHS section specifies the right-hand-side value for the rows. Any row unspecified in this section is considered to have an RHS value of 0. Missing the entire RHS section implies that all RHS values are 0.

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
RHS	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Blank	RHS name	Row name (e.g., rowi)	RHS value for row rowi	Row name (e.g., rowj)	RHS value for row rowj

Notes:

1. The rows that have an RHS element defined in this section need not be specified in the same order in which the rows were specified in the ROWS section. However, a row in the RHS section should be defined in the ROWS section.

2. It is possible to specify multiple RHS vectors, which are labeled by different RHS names. Normally, the first RHS vector encountered in the RHS section is used, and all other RHS vectors are discarded. All the elements of the selected RHS vector must be specified before other RHS vectors are introduced. Within a specific RHS vector, for a given row, duplicate assignments of RHS values are not allowed.

3. An RHS value assigned to the objective row is ignored by PROC OPTLP and PROC OPTMILP, while it is taken as a constant term of the objective function by PROC OPTQP.

4. A missing value in field4 or field6 is ignored. A data record with missing values in both field4 and field6 is ignored.

5. When a sequence of data records have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. If the value in field2 of the first data record in the section is missing, it means the name of the first vector is the missing value.

RANGES Section (Optional)

The RANGES section specifies the range of the RHS value for the constraint rows. With range specification, a row can be constrained from above and below.

For a constraint row c, if b is the RHS value and r is the range for this row, then the equivalent constraints are given in Table 17.1, depending on the type of row and the sign of r.

Table 17.1 Range Effect

Type of Row	Sign of r	Equivalent Constraints
G	$\pm$	$b \le c \le b + \lvert r \rvert$
L	$\pm$	$b - \lvert r \rvert \le c \le b$
E	$+$	$b \le c \le b + r$
E	$-$	$b + r \le c \le b$

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
RANGES	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Blank	Range name	Row name (e.g., rowi)	Range for RHS of row rowi	Row name (e.g., rowj)	Range for RHS of row rowj

Notes:

1. Range assignment for an objective row (i.e., **MAX**, **MIN**, or **N** row) is not allowed.
2. The rows that have a range element defined in this section need not be specified in the same order in which the rows were specified in the ROWS or RHS section. However, a row in the RANGES section should be defined in the ROWS section.
3. It is possible to specify multiple range vectors, which are labeled by different range names. Normally, the first range vector encountered in the RANGES section is used, and all other range vectors are discarded. All the elements in a range vector must be specified before other range vectors are introduced. Within the specific range vector, for a given range, duplicate assignments of range values are not allowed.
4. A missing value in field4 or field6 is ignored. A data record with missing values in both field4 and field6 is ignored.
5. When a sequence of data records have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. If the value in field2 of the first data record in the section is missing, it means the name of the first vector is the missing value.

BOUNDS Section (Optional)

The BOUNDS section specifies bounds for the columns.

- Indicator record:

Field1	**Field2**	**Field3**	**Field4**	**Field5**	**Field6**
BOUNDS	Blank	Blank	.	Blank	.

- Data record:

Field1	**Field2**	**Field3**	**Field4**	**Field5**	**Field6**
Bound type	Bound name	Column name	Bound for the column	Blank	Blank

Notes:

1. If you do not specify any bound for a column, then the upper bound is $+\infty$ for a continuous variable, and 1 for an integer variable, that is specified in the COLUMNS section. The lower bound is 0 by default.
2. General bound types include **LO**, **UP**, **FX**, **FR**, **MI**, and **PL**. Suppose the bound for a column identified in field3 is specified as *b* in field4. Table 17.2 explains the effects of different bound types.

Table 17.2 Bound Type Rules

Bound Type	Ignore *b*	Resultant Lower Bound	Resultant Upper Bound
LO	No	*b*	Unspecified
UP	No	Unspecified	*b*
FX	No	*b*	*b*
FR	Yes	$-\infty$	$+\infty$
MI	Yes	$-\infty$	Unspecified
PL	Yes	Unspecified	$+\infty$

If a bound (lower or upper) is not explicitly specified, then it takes the default values according to Note 1. There is one exception: if the upper bound is specified as a negative value ($b < 0$) and the lower bound is unspecified, then the lower bound is set to $-\infty$.

Mixed integer programming problems can specify integer variables in the BOUNDS section. Table 17.3 shows bound types defined for MIP.

Table 17.3 Bound Type Rules

Bound Type	Ignore *b*	Variable Type	Resultant Lower Bound	Resultant Upper Bound
BV	Yes	Binary	0	1
LI	No	Integer	*b*	$+\infty$
UI	No	Integer	Unspecified	*b*

3. The columns that have bounds do not need to be specified in the same order in which the columns were specified in the COLUMNS section. However, all columns in the BOUNDS section should be defined in the COLUMNS section.

4. It is possible to specify multiple bound vectors, which are labeled by different bound names. Normally, the first bound vector encountered in the BOUNDS section is used, and all other bound vectors are discarded. All the elements of the selected bound vector must be specified before other bound vectors are introduced.

5. When data records in a sequence have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. If the value in field2 of the first data record in the section is missing, it means the name of the first vector is the missing value.

6. Within a particular BOUNDS vector, for a given column, if a bound (lower or upper) is explicitly specified by the bound type rules listed in Table 17.2, any other specification is considered to be an error.

7. If the value in field1 is **LO**, **UP**, **FX**, **LI**, or **UI**, then a data record with a missing value in field4 is ignored.

BRANCH Section (Optional)

Sometimes you want to specify branching priorities or directions for integer variables to improve performance. Variables with higher priorities are branched on before variables with lower priorities. The branch direction is used to decide which branch to take first at each node. For more information, see the section "Branching Priorities" on page 646.

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
BRANCH	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Branch direction	Blank	First column name	First column priority	Second column name	Second column priority

Notes:

1. Valid directions include **UP** (up branch), **DN** (down branch), **RD** (rounding) and **CB** (closest bound). If field1 is blank, the solver automatically decides the direction.
2. If field4 is missing, then the name defined in field3 is ignored. Similarly, if field6 is missing, then the name defined in field5 is ignored.
3. The priority value in field4 and field6 must be nonnegative. Zero is the lowest priority and is also the default.

QSECTION Section (Optional)

The QSECTION section is needed only to describe quadratic programming problems. It specifies the coefficients of the quadratic terms in the objective function.

- Indicator record:

Field1	Field2	Field3	Field4	Field5	Field6
QSECTION or QUADOBJ	Blank	Blank	.	Blank	.

- Data record:

Field1	Field2	Field3	Field4	Field5	Field6
Blank	Column name	Column name	Coefficient in objective function	Blank	.

Notes:

1. The QSECTION section can appear only in PROC OPTQP and should not appear in PROC OPTLP. For PROC OPTLP, an error is reported when the submitted data set contains a QSECTION section. For PROC OPTQP, if the QSECTION section is not specified, or if there is no valid data record in the QSECTION section, then the LP solver is used to solve quadratic programming problems.
2. The variables field2 and field3 contain the names of the columns that form a quadratic term in the objective function. They must have been defined in the COLUMNS section. They need not refer to the same column. Zero entries should not be specified.
3. Duplicate entries of a quadratic term are not allowed. This means the combination of (field2, field3) must be unique, where (k, j) and (j, k) are considered to be the same combination.
4. If field4 of one data record is missing or takes a value of zero, then this data record is ignored.

ENDATA Section

The ENDATA section simply declares the end of all records. It contains only one indicator record, where field1 takes the value ENDATA and the values of the remaining variables are blank or missing.

Details: MPS-Format SAS Data Set

Converting an MPS/QPS-Format File: %MPS2SASD

As described in the section "Overview: MPS-Format SAS Data Set" on page 839, the MPS or QPS format is a standard file format for describing linear, integer, and quadratic programming problems. The MPS/QPS format is defined for plain text files, whereas in the SAS System it is more convenient to read data from SAS data sets. Therefore, a facility is required to convert MPS/QPS-format text files to MPS-format SAS data sets. The SAS macro %MPS2SASD serves this purpose.

In the MPS/QPS-format text file, a record refers to a single line of text that is divided into six fields. MPS/QPS files can be read and printed in both *fixed* and *free* format. In fixed MPS/QPS format, each field of a data record must occur in specific columns:

Field	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
Columns	2–3	5–12	15–22	25–36	40–47	50–61

In free format, fields of a record are separated by a space. Both fixed and free format have limitations. If users need to use row names or column names longer than 8 characters, then there is not enough space to hold them in fixed format. If users use a space as a part of a row name or column name, such as "ROW NAME", then free-format MPS format interprets this symbol as two fields, "ROW" and "NAME".

You can insert a comment record, denoted by an asterisk (*) in column 1, anywhere in an MPS/QPS file. Also, if a dollar sign ($) is the first character in field 3 or field 5 of any record, the information from that point to the end of the record is treated as a comment. All comments are ignored by the %MPS2SASD macro, described as follows.

%MPS2SASD Macro Parameters

%MPS2SASD (MPSFILE='infilename', OUTDATA=mpsdata, MAXLEN=n, FORMAT=FIXED/FREE);

MPSFILE=*'infilename'*
specifies the path and name of the input MPS-format file. The input file is a plain text file, normally with either an ".mps" extension for linear programming problems or a ".qps" extension for quadratic programming problems. This parameter is required; there is no default value.

OUTDATA=*mpsdata*
specifies the name of the output MPS-format SAS data set. This parameter is optional; the default value is mpsdata.

MAXLEN=*n*
specifies length of the variables field2, field3, and field5 in the output MPS-format SAS data set. This parameter is optional; the default value is 8.

FORMAT=*FIXED/FREE*
specifies the format of the input MPS file. Valid values can be either FIXED or FREE. This parameter is optional; the default value is the one, if any, specified by the flat file and FIXED otherwise.

Length of Variables

In an MPS-format SAS data set, normally the variables field2, field3, and field5 hold the names of the rows and columns. The length of these character variables is limited to the maximum size of a SAS character variable. This enables you to use sufficiently long names for the rows and columns in your model.

In a SAS data set generated by the %MPS2SASD macro, the length of the variables field2, field3, and field5 is fixed to be 8 ASCII characters by default. This length fits the fixed-format MPS/QPS file well since field 2, field 3, and field 5 are fixed at 8 characters. However, the free-format MPS/QPS files might have longer row names or longer column names. The %MPS2SASD macro provides a parameter **MAXLEN = n**. Using this parameter, you can set the variables field2, field3, and field5 to have a length of n characters in the output SAS data set.

The parameter MAXLEN works only when the given MPS file is in free format. For a fixed-format MPS file, this parameter is ignored and the length of field2, field3, and field5 is 8 characters by default.

Examples: MPS-Format SAS Data Set

Example 17.1: MPS-Format Data Set for a Product Mix Problem

Consider a simple product mix problem where a furniture company tries to find an optimal product mix of four items: a desk (x_1), a chair (x_2), a cabinet (x_3), and a bookcase (x_4). Each item is processed in a stamping department (STAMP), an assembly department (ASSEMB), and a finishing department (FINISH). The time each item requires in each department is given in the input data. Because of resource limitations, each department has an upper limit on the time available for processing. Furthermore, because of labor constraints, the assembly department must work at least 300 hours. Finally, marketing tells you not to make more than 75 chairs, to make at least 50 bookcases, and to find the range over which the selling price of a bookcase can vary without changing the optimal product mix.

This problem can be expressed as the following linear program:

$$
\begin{array}{rrcrl}
\max & 95x_1 + 41x_2 + 84x_3 + 76x_4 & & & \\
\text{subject to} & 3x_1 + 1.5x_2 + 2x_3 + 2x_4 & \leq & 800 & \text{(STAMP)} \\
& 10x_1 + 6x_2 + 8x_3 + 7x_4 & \leq & 1200 & \text{(ASSEMB)} \\
& 10x_1 + 6x_2 + 8x_3 + 7x_4 & \geq & 300 & \text{(ASSEMB)} \\
& 10x_1 + 8x_2 + 8x_3 + 7x_4 & \leq & 800 & \text{(FINISH)} \\
& x_2 & \leq & 75 & \\
& x_4 & \geq & 50 & \\
& x_i & \geq & 0 & i = 1, 2, 3
\end{array}
$$

The following DATA step saves the problem specification as an MPS-format SAS data set:

```
data prodmix;
   infile datalines;
   input field1 $ field2 $ field3$ field4 field5 $ field6;
   datalines;
NAME          .          PROD_MIX        .        .             .
ROWS          .             .            .        .             .
MAX         PROFIT          .            .        .             .
L           STAMP           .            .        .             .
L           ASSEMB          .            .        .             .
L           FINISH          .            .        .             .
COLUMNS       .             .            .        .             .
.           DESK        STAMP          3.0    ASSEMB          10
.           DESK        FINISH        10.0    PROFIT          95
.           CHAIR       STAMP          1.5    ASSEMB           6
.           CHAIR       FINISH         8.0    PROFIT          41
.           CABINET     STAMP          2.0    ASSEMB           8
.           CABINET     FINISH         8.0    PROFIT          84
.           BOOKCSE     STAMP          2.0    ASSEMB           7
.           BOOKCSE     FINISH         7.0    PROFIT          76
RHS           .             .            .        .             .
.           TIME        STAMP        800.0    ASSEMB        1200
.           TIME        FINISH       800.0       .             .
RANGES        .             .            .        .             .
```

```
.           T1          ASSEMB       900.0       .             .
BOUNDS        .             .           .         .             .
UP          BND         CHAIR         75.0       .             .
LO          BND         BOOKCSE       50.0       .             .
ENDATA        .             .           .         .             .
;
```

Example 17.2: Fixed-MPS-Format File

The following file, example_fix.mps, contains the data from Example 17.1 in the form of a fixed-MPS-format file. The indicator codes MAX and MIN are not available for objective rows in fixed MPS format, so the PROFIT row is specified as type N. Minimization is assumed for type-N rows; for a maximization objective, the objective coefficients must be replaced with values of the opposite sign.

```
* THIS IS AN EXAMPLE FOR FIXED MPS FORMAT.
NAME          PROD_MIX
ROWS
 N  PROFIT
 L  STAMP
 L  ASSEMB
 L  FINISH
COLUMNS
    DESK      STAMP          3.00000   ASSEMB        10.00000
    DESK      FINISH        10.00000   PROFIT       -95.00000
    CHAIR     STAMP          1.50000   ASSEMB         6.00000
    CHAIR     FINISH         8.00000   PROFIT       -41.00000
    CABINET   STAMP          2.00000   ASSEMB         8.00000
    CABINET   FINISH         8.00000   PROFIT       -84.00000
    BOOKCSE   STAMP          2.00000   ASSEMB         7.00000
    BOOKCSE   FINISH         7.00000   PROFIT       -76.00000
RHS
    TIME      STAMP        800.00000   ASSEMB       1200.0000
    TIME      FINISH       800.00000
RANGES
    T1        ASSEMB       900.00000
BOUNDS
 UP BND       CHAIR         75.00000
 LO BND       BOOKCSE       50.00000
ENDATA
```

Example 17.3: Free-MPS-Format File

In free format, fields in data records other than the first record have no predefined positions. They can be written anywhere except column 1, with each field separated from the next by one or more blanks (a tab cannot be used as a field separator). However, the fields must appear in the same sequence as in the

fixed format. The following file, example_free.mps, is an example. It describes the same problem as in Example 17.2.

```
* THIS IS AN EXAMPLE FOR FREE MPS FORMAT.
NAME          PROD_MIX  FREE
ROWS
 N  PROFIT
    L  STAMP
    L  ASSEMB
    L  FINISH
COLUMNS
 DESK       STAMP    3.00000   ASSEMB      10.00000
 DESK       FINISH   10.00000    PROFIT   -95.00000
    CHAIR   STAMP    1.50000   ASSEMB       6.00000
    CHAIR   FINISH   8.00000     PROFIT   -41.00000
 CABINET    STAMP    2.00000   ASSEMB       8.00000
 CABINET    FINISH   8.00000     PROFIT   -84.00000
    BOOKCSE STAMP    2.00000   ASSEMB       7.00000
    BOOKCSE FINISH   7.00000     PROFIT   -76.00000
RHS
    TIME STAMP   800.00000  ASSEMB 1200.0000
    TIME FINISH  800.00000
RANGES
    T1   ASSEMB   900.00000
BOUNDS
 UP BND     CHAIR 75.00000
 LO BND   BOOKCSE 50.00000
ENDATA
```

Example 17.4: Using the %MPS2SASD Macro

We illustrate the use of the %MPS2SASD macro in this example, assuming the files example_fix.mps and example_free.mps are in your current SAS working directory.

The MPS2SASD macro function has one required parameter, MPSFILE= ‘infilename’, which specifies the path and name of the MPS/QPS-format file. With this single parameter, the macro reads the file, converts the records, and saves the conversion to the default MPS-format SAS data set MPSDATA.

Running the following statements converts the fixed-format MPS file shown in Example 17.2 to the MPS-format SAS data set MPSDATA:

```
%mps2sasd(mpsfile='example_fix.mps');
proc print data=mpsdata;
run;
```

Output 17.4.1 displays the MPS-format SAS data set MPSDATA.

Output 17.4.1 The MPS-Format SAS Data Set MPSDATA

Obs	field1	field2	field3	field4	field5	field6
1	NAME		PROD_MIX	.		.
2	ROWS			.		.
3	N	PROFIT		.		.
4	L	STAMP		.		.
5	L	ASSEMB		.		.
6	L	FINISH		.		.
7	COLUMNS			.		.
8		DESK	STAMP	3.0	ASSEMB	10
9		DESK	FINISH	10.0	PROFIT	-95
10		CHAIR	STAMP	1.5	ASSEMB	6
11		CHAIR	FINISH	8.0	PROFIT	-41
12		CABINET	STAMP	2.0	ASSEMB	8
13		CABINET	FINISH	8.0	PROFIT	-84
14		BOOKCSE	STAMP	2.0	ASSEMB	7
15		BOOKCSE	FINISH	7.0	PROFIT	-76
16	RHS			.		.
17		TIME	STAMP	800.0	ASSEMB	1200
18		TIME	FINISH	800.0		.
19	RANGES			.		.
20		T1	ASSEMB	900.0		.
21	BOUNDS			.		.
22	UP	BND	CHAIR	75.0		.
23	LO	BND	BOOKCSE	50.0		.
24	ENDATA			.		.

Running the following statement converts the free-format MPS file shown in Example 17.3 to the MPS-format SAS data set MPSDATA:

```
%mps2sasd(mpsfile='example_free.mps');
```

The data set is identical to the one shown in Output 17.4.1.

In the following statement, when the free-format MPS file is converted, the length of the variables field2, field3, and field5 in the SAS data set MPSDATA is explicitly set to 10 characters:

```
%mps2sasd(mpsfile='example_free.mps', maxlen=10, format=free);
```

If you want to save the converted data to a SAS data set other than the default data set MPSDATA, you can use the parameter OUTDATA= mpsdata. The following statement reads data from the file example_fix.mps and writes the converted data to the data set PRODMIX:

```
%mps2sasd(mpsfile='example_fix.mps', outdata=PRODMIX);
```

References

IBM (1988). *Mathematical Programming System Extended/370 (MPSX/370) Version 2 Program Reference Manual.* Vol. SH19-6553-0. Armonk, NY: IBM.

Maros, I., and Mészáros, C. (1999). "A Repository of Convex Quadratic Programming Problems." *Optimization Methods and Software* 11–12:671–681.

Murtagh, B. A. (1981). *Advanced Linear Programming: Computation and Practice*. New York: McGraw-Hill.

Subject Index

active nodes
    OPTMILP procedure, 643
    OPTMODEL procedure, MILP solver, 337
active-set method
    overview, 510
active-set primal-dual algorithm, 504
ACTIVITY variable
    DUALOUT= data set, 586, 642, 698
aggregation operators
    OPTMODEL procedure, 96
algorithm, 260, 739
assignment strategy, 211
    variable, 200

backtracking search, 209
Bard function, 167
basis, 264, 580
BLOCK variable
    BLOCKS= data set, 741
block-angular structure
    decomposition algorithm, 717, 769, 779, 784
block-diagonal structure
    decomposition algorithm, 717, 737, 743, 762
block-diagonal structure in distributed mode
    decomposition algorithm, 766
blocks
    decomposition algorithm, 717, 741
BLOCKS= data set
    blocks, 741
    DECOMP statement, 741
    decomposition algorithm, 741
    variables, 741
branch-and-bound
    control options, 338, 644
branch-and-price
    decomposition algorithm, 742
branching priorities
    OPTMILP procedure, 646
    OPTMODEL procedure, MILP solver, 339
branching priority
    MPS-format SAS data set, 847
branching variable
    OPTMILP procedure, 643
    OPTMODEL procedure, MILP solver, 337

CLOSEFILE statement
    OPTMODEL procedure, 122
CLP solver
    assignment strategy, 211
    consistency techniques, 210
    details, 209
    getting started, 194
    overview, 194, 209
    selection strategy, 211
column generation
    decomposition algorithm, 742
columns, 117
complementarity
    OPTMODEL procedure, 114
concurrent LP, 272, 593
consistency techniques, 210
constrained optimization
    overview, 507
constraint bodies
    OPTMODEL procedure, 132
constraint declaration
    OPTMODEL procedure, 44
constraint programming
    finite domain, 210
constraint propagation, 209
constraint satisfaction problem (CSP), 194
    backtracking search, 209
    constraint propagation, 209
    definition, 194
    solving techniques, 209
    standard CSP, 209
constraint status
    LP solver, 273
constraints
    OPTMODEL procedure, 30, 130
control flow
    OPTMODEL procedure, 121
conversions
    OPTMODEL procedure, 155
converting MPS-format file
    examples, 852
    MPS2SASD macro, 848
covariance matrix, 500
coverage
    decomposition algorithm, 717, 747, 759, 788
cutting planes
    OPTMILP procedure, 646
    OPTMODEL procedure, MILP solver, 340

Dantzig-Wolfe method
    decomposition algorithm, 742
data, 576, 630

data set input/output
    OPTMODEL procedure, 117
declaration statements
    OPTMODEL procedure, 44
DECOMP statement
    BLOCKS= data set, 741
    definitions of BLOCKS= data set variables, 741
decomposition algorithm
    block-angular structure, 717, 769, 779, 784
    block-diagonal structure, 717, 737, 743, 762
    block-diagonal structure in distributed mode, 766
    blocks, 717, 741
    BLOCKS= data set, 741
    branch-and-price, 742
    column generation, 742
    coverage, 717, 747, 759, 788
    Dantzig-Wolfe method, 742
    decomposition algorithm, 742
    details, 741
    examples, 749
    introductory example, 718
    Lagrangian decomposition, 779, 780
    LP solver, 265
    master problem, 716, 717, 742
    MILP solver, 336
    OPTLP procedure, 582
    OPTMILP procedure, 640
    overview, 716
    pricing out variables, 742
    relaxation, 716, 759
    Ryan-Foster branching, 744
    separable region, 717
    set covering, 746
    set packing, 747
    set partitioning, 744
    subproblem, 716, 718, 741, 742
decomposition algorithm examples
    ATM cash management in distributed mode, 808
    ATM cash management in single-machine mode, 797
    bin packing problem, 773
    block-angular structure, 769
    block-diagonal structure, 762
    block-diagonal structure in distributed mode, 766
    generalized assignment problem, 755
    kidney donor exchange, 811
    multicommodity flow, 749
    resource allocation, 778
    vehicle routing problem, 791
deterministic solution results, 22
distributed mode
    OPTLP procedure, 582
    OPTMILP option tuner, 824
    OPTMILP procedure, 640
domain, 194
    distribution strategy, 210
dual value
    OPTMODEL procedure, 138
DUALIN= data set
    OPTLP procedure, 583
    variables, 583
dualization, 261, 578
DUALOUT= data set
    OPTLP procedure, 585, 586
    OPTMILP procedure, 642
    OPTQP procedure, 696–698
    variables, 585, 586, 642, 696–698

examples, *see* OPTLP examples, *see* OPTMODEL examples, *see* OPTQP examples, 194
    alphabet blocks problem, 221
    converting MPS-format file, 852
    Eight Queens problem, 197
    fixed MPS-format file, 851
    free MPS-format file, 851
    logic-based puzzles, 214
    Magic Square problem, 219
    MPS-format SAS data set, 850
    Pi Day sudoku problem, 216
    scene allocation problem, 233
    Send More Money problem, 195
    sudoku problem, 214
    work-shift scheduling problem, 224
expressions
    OPTMODEL procedure, 99

facility location
    MILP solver examples, 357
FCMP routines
    OPTMODEL procedure, 156
feasibility tolerance, 262, 578
feasible region, 115
    OPTMODEL procedure, 30
feasible solution, 115
    OPTMODEL procedure, 30
FILE statement
    OPTMODEL procedure, 122
finite-domain constraint programming, 210
first-order necessary conditions
    local minimum, 116
fixed MPS-format file
    examples, 851
FOR statement
    OPTMODEL procedure, 121
formatted output
    OPTMODEL procedure, 121
free MPS-format file
    examples, 851

function expressions
OPTMODEL procedure, 102
functional summary
OPTMODEL procedure, 38

global solution, 115
graph theory and network analysis, 376

identifier expressions
OPTMODEL procedure, 102
IF expression, 106
IIS option
OPTLP procedure, 597
OPTMODEL procedure, LP solver, 274
OPTQP procedure, 704
impure functions
OPTMODEL procedure, 97
index sets, 96
implicit set slicing, 160
index-set-item, 104
OPTMODEL procedure, 103
INITIAL variable
OPTIONVALUES= data set, 828
integer variables
OPTMODEL procedure, 137
interior point algorithm
overview, 508
interior point primal-dual algorithm, 504
intermediate variable, 189
irreducible infeasible set
OPTLP procedure, 597
OPTMODEL procedure, LP solver, 274
iteration log
crossover algorithm, 271, 592
interior point algorithm, 271, 592
LP solver, 269–271
network simplex algorithm, 270, 591
OPTLP procedure, 590–592
primal and dual simplex algorithms, 269, 590

Karush-Kuhn-Tucker conditions, 116
key columns, 117, 119
key set, 64

Lagrange multipliers, 116
Lagrangian decomposition
decomposition algorithm, 779, 780
Lagrangian function, 116
LBOUND variable
PRIMALOUT= data set, 584, 642, 696
linear programming, *see also* OPTMODEL procedure, *see also* OPTLP procedure
OPTMODEL procedure, 30
list form
PRINT statement, 77

local minimum
first-order necessary conditions, 116
second-order necessary conditions, 116
local solution, 115
look-ahead schemas, 210
look-back schemas, 210
LP solver
concurrent LP, 272
constraint status, 273
iteration log, 269–271
problem statistics, 272
variable status, 273
LP solver examples
diet problem, 278
finding an irreducible infeasible set, 292
generalized networks, 303
maximum flow, 307
production, inventory, distribution, 310
shortest path, 318
two-person zero-sum game, 289
using the network simplex algorithm, 295
_L_RHS_ variable
DUALOUT= data set, 585, 642, 697

%MPS2SASD
MPS2SAD, 689
MPS2SASD, 692
macro variable
OROPTMODEL, 161, 211, 275, 458, 518, 558
macro variable
OROPTMILP, 654
OROPTLP
OROPTLP, 598
OROPTMODEL
OROPTMODEL, 344
OROPTQP
OROPTQP, 705
master problem
decomposition algorithm, 716, 717, 742
matrix form
PRINT statement, 78
memory limit, 23
method, 730
migration to PROC OPTMODEL
from PROC NETFLOW, 303, 307, 310, 318
from PROC NLP, 163
MILP solver
problem statistics, 343
MILP solver examples
branching priorities, 365
facility location, 357, 666
miplib, 662
multicommodity problems, 351
scheduling, 347, 674

simple integer linear program, 657
traveling salesman problem, 368
model building
PROC OPTMODEL, 15
model update
OPTMODEL procedure, 145
Moore-Penrose conditions, 515
MPS format, 86
MPS-format file, 848
MPS-format SAS data set, 839
bound type, 845
branching priority, 847
converting MPS-format file, 852
examples, 850
length of variables, 849
range, 844
row type, 841
sections, 841
variables, 840
MPS2SASD macro
converting MPS-format file, 848
multicommodity problems
MILP solver examples, 351
multiple solutions
OPTMODEL procedure, 151
multiple subproblems
OPTMODEL procedure, 150
multithreaded parallel computing, 21
multithreading
OPTLP procedure, 582
OPTMILP procedure, 640
OPTQP procedure, 695

NAME variable
PROBLEMS= data set, 828
network solver
overview, 376
NLP solver
covariance matrix, 500, 514
eigenvalue tolerance, 501
singularity criterion, 500, 501
solver termination messages, 518
NLP solver examples
finding an irreducible infeasible set, 538
maximum likelihood Weibull estimation, 536
solving highly nonlinear optimization problems, 521
solving large-scale NLP problems, 528
solving NLP problems that have several local minima, 530
solving NLP problems with range constraints, 525
solving unconstrained and bound-constrained optimization problems, 523
node selection
OPTMILP procedure, 645
OPTMODEL procedure, MILP solver, 338
nondeterministic solution results, 22
numerical difficulties, 24

OBJCOEF variable
PRIMALOUT= data set, 642
VAR variable
PRIMALOUT= data set, 584, 696
objective declarations
OPTMODEL procedure, 32, 47
objective functions
OPTMODEL procedure, 30, 32, 47
objective value
OPTMODEL procedure, 30
objectives
OPTMODEL procedure, 32
_OBJ_ID_ variable
DUALOUT= data set, 585, 642, 696
PRIMALOUT= data set, 583, 641, 695
OBJSENSE variable
PROBLEMS= data set, 828
TUNEROUT= data set, 828
ODS table names
OPTLP procedure, 593
OPTMILP procedure, 650
OPTMODEL procedure, 124
OPTQP procedure, 700
PERFORMANCE statement, 23
ODS variable names
OPTMODEL procedure, 125
online documentation, 8
operators
OPTMODEL procedure, 100
optimal solution
OPTMODEL procedure, 30
optimal value
OPTMODEL procedure, 30
optimality conditions
OPTMODEL procedure, 114
optimization
introduction, 11
optimization modeling language, 28
optimization variable
OPTMODEL procedure, 30
optimization variables
OPTMODEL procedure, 32
option tuner, *see* OPTMILP option tuner
OPTION variable
OPTIONVALUES= data set, 828
OPTIONVALUES= data set
INITIAL variable, 828
OPTION variable, 828
OPTMILP option tuner, 828

VALUES variable, 828
variables, 828
OPTLP examples
diet optimization problem, 605
finding an irreducible infeasible set, 615
oil refinery problem, 600
reoptimizing after adding a new constraint, 612
reoptimizing after modifying the objective function, 607
reoptimizing after modifying the right-hand side, 610
using the interior point algorithm, 604
using the network simplex algorithm, 620
OPTLP procedure
algorithm, 577
basis, 580
concurrent LP, 593
crossover, 581
data, 576
decomposition algorithm, 582
definitions of DUALIN= data set variables, 583
definitions of DUALOUT= data set variables, 585, 586
definitions of DUALOUT=data set variables, 585, 586
definitions of PRIMALIN data set variables, 583
definitions of PRIMALIN= data set variables, 583
definitions of PRIMALOUT= data set variables, 583–585
distributed mode, 582
dual infeasibility, 581
DUALIN= data set, 583
duality gap, 581
dualization, 578
DUALOUT= data set, 585, 586
feasibility tolerance, 578
functional summary, 575
IIS option, 597
interior point algorithm, 588
introductory example, 572
iteration log, 590–592
memory limit, 23
multithreaded parallel computing, 21
multithreading, 582
network simplex algorithm, 588
numerical difficulties, 24
ODS table names, 593
OROPTLP macro variable, 598
preprocessing, 578
presolver, 578
pricing, 580
primal infeasibility, 582
PRIMALIN= data set, 583
PRIMALOUT= data set, 583–585
problem statistics, 596
queue size, 581
random seed, 581
scaling, 581
single-machine mode, 582
OPTMILP option tuner, 640
distributed mode, 824
examples, 822, 833
functional summary, 824
OPTIONVALUES= data set, 828
overview, 821
PROBLEMS= data set, 828
single-machine mode, 824
syntax, 824
TUNEROUT= data set, 828
OPTMILP option tuner examples
default tuning options, 833
distributed mode, 838
multiple problems, 835
single problem, 833
single-machine mode, 833, 835
user-defined tuning options, 835
OPTMILP procedure
active nodes, 643
branch-and-bound, 644
branching priorities, 646
branching variable, 643
cutting planes, 646
data, 630
decomposition algorithm, 640
definitions of DUALOUT= data set variables, 642
definitions of DUALOUT=data set variables, 642
definitions of PRIMALIN= data set variables, 641
definitions of PRIMALOUT= data set variables, 641, 642
distributed mode, 640
DUALOUT= data set, 642
functional summary, 629
introductory example, 626
memory limit, 23
multithreaded parallel computing, 21
multithreading, 640
node selection, 645
numerical difficulties, 24
ODS table names, 650
OROPTMILP macro variable, 654
presolve, 646
PRIMALIN= data set, 641
PRIMALOUT= data set, 641, 642
probing, 646
problem statistics, 654
random seed, 634
single-machine mode, 640
variable selection, 645

OPTMODEL examples
chemical equilibrium, 188
matrix square root, 166
model construction, 169
multiple subproblems, 174
reading from and creating a data set, 167
set manipulation, 173
sparse modeling, 182
SUBMIT statement, 178
traveling salesman problem, 178
OPTMODEL expression extensions, 104
aggregation expression, 108
OPTMODEL procedure
aggregation operators, 96
CLOSEFILE statement, 122
complementarity, 114
constraint bodies, 132
constraints, 130
control flow, 121
conversions, 155
data set input/output, 117
declaration statements, 44
dual value, 138
dualization, 261
expressions, 99
FCMP routines, 156
feasible region, 115
feasible solution, 115
FILE statement, 122
first-order necessary conditions, 116
FOR statement, 121
formatted output, 121
function expressions, 102
functional summary, 38
global solution, 115
identifier expressions, 102
impure functions, 97
index sets, 103
integer variables, 137
Karush-Kuhn-Tucker conditions, 116
Lagrange multipliers, 116
Lagrangian function, 116
local solution, 115
macro variable _OROPTMODEL_, 161
memory limit, 23
model update, 145
multiple solutions, 151
multiple subproblems, 150
multithreaded parallel computing, 21
numerical difficulties, 24
objective declarations, 32, 47
ODS table names, 124
ODS variable names, 125
operators, 100
optimality conditions, 114
optimization variables, 32
options classified by function, 38
_OROPTMODEL_NUM_KEYS_ parameter, 161
_OROPTMODEL_NUM_ parameter, 161
_OROPTMODEL_STR_KEYS_ parameter, 161
_OROPTMODEL_STR_ parameter, 161
overview, 28
parameters, 47, 95
presolver, 145
primary expressions, 101
PRINT statement, 122
programming statements, 53
PUT statement, 121
range constraints, 140
reduced costs, 144
RESET OPTIONS statement, 153
second-order necessary conditions, 116
second-order sufficient conditions, 117
_SOLUTION_STATUS_ parameter, 161
STATUS parameter, 161
strict local solution, 115
suffix names, 132, 134
table of syntax elements, 38
threaded and distributed processing, 160
variable declaration, 32, 52
OPTMODEL procedure, CLP solver
macro variable _OROPTMODEL_, 211
OPTMODEL procedure, DECOMP algorithm
method, 730
OPTMODEL procedure, DECOMP_SUBPROB algorithm
algorithm, 739
OPTMODEL procedure, LP solver
algorithm2, 261
basis, 264
feasibility tolerance, 262
functional summary, 259
IIS option, 274
introductory example, 256
macro variable _OROPTMODEL_, 275
network simplex algorithm, 266
preprocessing, 261
presolver, 261
pricing, 264
queue size, 264
scaling, 264
solver, 260
OPTMODEL procedure, MILP solver
active nodes, 337
branch-and-bound, 338
branching priorities, 339
branching variable, 337
cutting planes, 340

functional summary, 325
introductory example, 324
node selection, 338
OROPTMODEL macro variable, 344
presolve, 340
probing, 340
variable selection, 339
OPTMODEL procedure, network solver
macro variable _OROPTMODEL_, 458
OPTMODEL procedure, NLP solver
details, 506
functional summary, 499
introductory examples, 489
macro variable _OROPTMODEL_, 518
solver, 504
technique, 504
OPTMODEL procedure, QP solver
functional summary, 551
macro variable _OROPTMODEL_, 558
OPTQP examples
covariance matrix, 709
data fitting, 707
estimation, 707
linear least squares, 707
Markowitz model, 709
portfolio optimization, 709
portfolio selection with transactions, 712
short-sell, 711
OPTQP procedure
output data sets, 695
definitions of DUALOUT= data set variables, 696, 697
definitions of DUALOUT=data set variables, 697, 698
definitions of PRIMALOUT= data set variables, 695, 696
dual infeasibility, 694
duality gap, 694
DUALOUT= data set, 696–698
examples, 706
functional summary, 692
IIS option, 704
interior point algorithm overview, 698
iteration log, 693
memory limit, 23
%MPS2SASD macro, 689, 692
multithreaded parallel computing, 21
multithreading, 695
numerical difficulties, 24
ODS table names, 700
OROPTQP macro variable, 705
overview, 685
primal infeasibility, 694
PRIMALOUT= data set, 695, 696
problem statistics, 703
OROPTMODEL macro variable, 211, 275, 458, 518, 558
_OROPTMODEL_NUM_KEYS_ parameter
OPTMODEL procedure, 161
_OROPTMODEL_NUM_ parameter
OPTMODEL procedure, 161
_OROPTMODEL_STR_KEYS_ parameter
OPTMODEL procedure, 161
_OROPTMODEL_STR_ parameter
OPTMODEL procedure, 161
overview
network solver, 376
optimization, 11
OPTMODEL procedure, 28
OPTQP procedure, 685

parallel processing
parallel processing, 743
parameters, 98
initialization, 99
OPTMODEL procedure, 47, 95
_OROPTMODEL_NUM_KEYS_ parameter, 161
_OROPTMODEL_NUM_ parameter, 161
_OROPTMODEL_STR_KEYS_ parameter, 161
_OROPTMODEL_STR_ parameter, 161
parameter declarations, 47
parameter options, 48
_SOLUTION_STATUS_ parameter, 161
STATUS parameter, 161
PDIGITS= option, 123
PERFORMANCE statement
ODS table names, 23
positive semidefinite matrix, 546, 686
predicates, 202
presolve
OPTMILP procedure, 646
OPTMODEL procedure, MILP solver, 340
presolver, 261, 578
pricing, 264, 580
pricing out variables
decomposition algorithm, 742
PRIMALIN= data set
OPTLP procedure, 583
OPTMILP procedure, 641
variables, 583, 641
PRIMALOUT= data set
OPTLP procedure, 583–585
OPTMILP procedure, 641, 642
OPTQP procedure, 695, 696
variables, 583–585, 641, 642, 695, 696
primary expressions
OPTMODEL procedure, 101
PRINT statement

list form, 77
matrix form, 78
OPTMODEL procedure, 122
probing
OPTMILP procedure, 646
OPTMODEL procedure, MILP solver, 340
PROBLEM variable
TUNEROUT= data set, 828
PROBLEMS= data set
NAME variable, 828
OBJSENSE variable, 828
OPTMILP option tuner, 828
variables, 828
PROC OPTMODEL
model building, 15
programming statements
control, 53
general, 53
input/output, 53
looping, 53
model, 53
OPTMODEL procedure, 53
PUT statement
OPTMODEL procedure, 121
PWIDTH= option, 123

QP Solver
examples, 559
interior point algorithm overview, 554
iteration log, 556
QP solver
IIS, 557
problem statistics, 556
QP solver examples
covariance matrix, 562
data fitting, 559
estimation, 559
linear least squares, 559
Markowitz model, 562
portfolio optimization, 562
portfolio selection with transactions, 566
short-sell, 565
QPS format, 88
QPS format file, 848
quadratic programming
quadratic matrix, 546, 686
queue size, 264, 581

random seed, 265, 330, 581, 634
range constraints
OPTMODEL procedure, 140
RANK variable
TUNEROUT= data set, 829
_R_COST_ variable
PRIMALOUT= data set, 585
READ DATA statement
trim option, 83
reduced costs
OPTMODEL procedure, 144
relaxation
decomposition algorithm, 716, 759
RESET OPTIONS statement
OPTMODEL procedure, 153
RHS variable
DUALOUT= data set, 585, 642, 697
_RHS_ID_ variable
DUALOUT= data set, 585, 642, 697
PRIMALOUT= data set, 584, 641, 695
ROW variable
BLOCKS= data set, 741
DUALIN= data set, 583
DUALOUT= data set, 585, 642, 697
Ryan-Foster branching
decomposition algorithm, 744

satisfiability problem (SAT), 209
scalar types, 47, 97
scaling, 264, 581
scheduling
MILP solver examples, 347
scheduling mode
CLP procedure, 211
second-order necessary conditions, 116
local minimum, 116
second-order sufficient conditions, 117
strict local minimum, 117
selection strategy, 211
MINR, 211
separable region
decomposition algorithm, 717
set covering
decomposition algorithm, 746
set packing
decomposition algorithm, 747
set partitioning
decomposition algorithm, 744
set types, 47
single-machine mode
OPTLP procedure, 582
OPTMILP option tuner, 824
OPTMILP procedure, 640
singularity, 514
absolute singularity criterion, 500
relative singularity criterion, 501
_SOLUTION_STATUS_ parameter
OPTMODEL procedure, 161
SOLVE WITH LP statement
crossover, 265

dual infeasibility, 265
duality gap, 265
primal infeasibility, 265
SOLVE WITH QP statement
dual infeasibility, 553
duality gap, 552
primal infeasibility, 553
standard CSP, 209
STATUS parameter
OPTMODEL procedure, 161
STATUS variable
DUALIN= data set, 583
DUALOUT= data set, 586, 697
PRIMALIN= data set, 583
PRIMALOUT= data set, 584, 696
strict local minimum
second-order sufficient conditions, 117
strict local solution, 115
subproblem
decomposition algorithm, 716, 718, 741, 742
suffix names
OPTMODEL procedure, 132, 134
suffixes, 119, 134

threaded and distributed processing
OPTMODEL procedure, 160
traveling salesman problem
MILP solver examples, 368
trim option
READ DATA statement, 83
TUNEROUT= data set
OBJSENSE variable, 828
option configurations, 829
OPTMILP option tuner, 828
PROBLEM variable, 828
RANK variable, 829
solution information, 829
variables, 828
tuples, 97
TYPE variable
DUALOUT= data set, 585, 642, 697
PRIMALOUT= data set, 584, 641, 696

UBOUND variable
PRIMALOUT= data set, 584, 642, 696
unconstrained optimization
OPTMODEL procedure, 30
_U_RHS_ variable
DUALOUT= data set, 586, 642, 697

VALUE variable
DUALOUT= data set, 586, 697
PRIMALIN= data set, 641
PRIMALOUT= data set, 584, 642, 696
VALUES variable
OPTIONVALUES= data set, 828
VAR variable
PRIMALIN= data set, 583, 641
PRIMALOUT= data set, 584, 641, 696
variable declaration
OPTMODEL procedure, 32, 52
variable selection, 210
OPTMILP procedure, 645
OPTMODEL procedure, MILP solver, 339
variable status
LP solver, 273

Syntax Index

ABSOBJGAP= option
DECOMP statement, 727
PROC OPTMILP statement, 631
SOLVE WITH MILP statement, 328
ABSOBJGAP= suboption
TSP= option, 394
ALGORITHM2= option
PROC OPTLP statement, 577
SOLVE WITH LP statement, 261
ALGORITHM= option
DECOMP_SUBPROB statement, 739
PROC OPTLP statement, 577
SOLVE WITH LP statement, 260
SOLVE WITH NLP statement, 504
ALGORITHM= suboption
CONCOMP= option, 391
ALLDIFF predicate, 203
AND aggregation expression
OPTMODEL expression extensions, 104
ARTPOINTS= suboption
OUT= option, 388
ASINGULAR= option
SOLVE WITH NLP statement, 500, 514
assignment statement
OPTMODEL procedure, 53
ASSIGNMENTS= suboption
OUT= option, 389

BASIS= option
PROC OPTLP statement, 580
SOLVE WITH LP statement, 264
BICONCOMP option
algorithm options, 390
BICONCOMP= suboption
OUT= option, 389
BLOCKS= option
DECOMP statement, 727

CALL statement
OPTMODEL procedure, 54
CARD function
OPTMODEL expression extensions, 105
CDIGITS= option
PROC OPTMODEL statement, 40
CLIQUE= option
algorithm options, 391
CLIQUES= suboption
OUT= option, 389
CLOSEFILE statement
OPTMODEL procedure, 54
CLP solver, 199
COFOR statement
OPTMODEL procedure, 55
COL keyword
CREATE DATA statement, 62, 64
READ DATA statement, 84
COMPRESSFREQ= option
DECOMP statement, 727
CONCOMP= option
algorithm options, 391
CONCOMP= suboption
OUT= option, 389
CONFLICTSEARCH= option
PROC OPTMILP statement, 635
SOLVE WITH MILP statement, 332
CONFLICTSEARCH= suboption
TSP= option, 394
CONSTRAINT option
EXPAND statement, 70
CONSTRAINT statement
OPTMODEL procedure, 44
CONTINUE statement
OPTMODEL procedure, 61
COV= option
SOLVE WITH NLP statement, 500
COVEST=() option
SOLVE WITH NLP statement, 513
COVEST=() option
SOLVE WITH NLP statement, 500
COVOUT= option
SOLVE WITH NLP statement, 501
COVSING= option
SOLVE WITH NLP statement, 501, 515
CREATE DATA statement
COL keyword, 62, 64
OPTMODEL procedure, 61
CROSS expression
OPTMODEL expression extensions, 105
CROSSOVER= option
PROC OPTLP statement, 581
SOLVE WITH LP statement, 265
CUTCLIQUE= option
PROC OPTMILP statement, 638
SOLVE WITH MILP statement, 334
CUTFLOWCOVER= option
PROC OPTMILP statement, 638
SOLVE WITH MILP statement, 335

CUTFLOWPATH= option
PROC OPTMILP statement, 638
SOLVE WITH MILP statement, 335
CUTGOMORY= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 335
CUTGUB= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 335
CUTIMPLIED= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 335
CUTKNAPSACK= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 335
CUTLAP= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 335
CUTMILIFTED= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 335
CUTMIR= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 335
CUTMULTICOMMODITY= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 335
CUTOFF= option
PROC OPTMILP statement, 632
SOLVE WITH MILP statement, 328
CUTOFF= suboption
TSP= option, 395
CUTS= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 336
CUTSETS= suboption
OUT= option, 389
CUTSFACTOR= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 336
CUTSTRATEGY= option
PROC OPTMILP statement, 639
SOLVE WITH MILP statement, 336
CUTSTRATEGY= suboption
TSP= option, 395
CUTZEROHALF= option
PROC OPTMILP statement, 640
SOLVE WITH MILP statement, 336
CYCLE= option
algorithm options, 391
CYCLES= suboption
OUT= option, 389

DATA= option
PROC OPTLP statement, 576
PROC OPTMILP statement, 630
PROC OPTQP statement, 692
DECOMP_MASTER_IP=() option
SOLVE WITH MILP statement, 336
DECOMP_MASTER_IP statement
DECOMP option, 734
OPTMILP procedure, 640
DECOMP_MASTER=() option
SOLVE WITH LP statement, 265
SOLVE WITH MILP statement, 336
DECOMP_MASTER statement
DECOMP option, 732
OPTLP procedure, 582
OPTMILP procedure, 640
DECOMP option
DECOMP_MASTER_IP statement, 734
DECOMP_MASTER statement, 732
DECOMP statement, 726
DECOMP_SUBPROB statement, 736
syntax, 721
DECOMP=() option
SOLVE WITH LP statement, 265
SOLVE WITH MILP statement, 336
DECOMP statement
ABSOBJGAP= option, 727
BLOCKS= option, 727
COMPRESSFREQ= option, 727
DECOMP option, 726
HYBRID= option, 727
INITVARS= option, 728
LOGFREQ= option, 728
LOGLEVEL= option, 728
MASTER_IP_BEG= option, 729
MASTER_IP_END= option, 729
MASTER_IP_FREQ= option, 730
MAXBLOCKS= option, 730
MAXCOLSPASS= option, 730
MAXITER= option, 730
MAXTIME= option, 730
METHOD= option, 730
NBLOCKS= option, 731
NTHREADS= option, 731
OPTLP procedure, 582
OPTMILP procedure, 640
RELOBJGAP= option, 732
DECOMP_SUBPROB=() option
SOLVE WITH LP statement, 266
SOLVE WITH MILP statement, 336
DECOMP_SUBPROB statement
DECOMP option, 736
OPTLP procedure, 582
OPTMILP procedure, 640
DECOMP_MASTER statement

INITPRESOLVER= option, 733
NTHREADS= option, 733
DECOMP_MASTER_IP statement
NTHREADS= option, 735
PRIMALIN= option, 736
DECOMP_SUBPROB statement
ALGORITHM= option, 739
INITPRESOLVER= option, 740
NTHREADS= option, 740
PRIMALIN= option, 740
SOL= option, 739
SOLVER= option, 739
DETAILS option
PERFORMANCE statement, 22
DIFF expression
OPTMODEL expression extensions, 105
DO statement
END keyword, 66
OPTMODEL procedure, 66
DO statement, iterative
END keyword, 66
OPTMODEL procedure, 66
UNTIL keyword, 66
WHILE keyword, 66
DO UNTIL statement
END keyword, 68
OPTMODEL procedure, 68
DO WHILE statement
END keyword, 68
OPTMODEL procedure, 68
DROP statement
OPTMODEL procedure, 69
DUALIN= option
PROC OPTLP statement, 576
DUALIZE= option
PROC OPTLP statement, 578
SOLVE WITH LP statement, 261
DUALOUT= option
PROC OPTLP statement, 576
PROC OPTMILP statement, 631
DUALOUT=option
PROC OPTQP statement, 692

ELEMENT predicate, 204
ELSE keyword
IF statement, 74
EMPHASIS= option
PROC OPTMILP statement, 632
SOLVE WITH MILP statement, 328
EMPHASIS= suboption
TSP= option, 395
END keyword
DO statement, 66
DO statement, iterative, 66
DO UNTIL statement, 68
DO WHILE statement, 68
ERRORLIMIT= option
PROC OPTMODEL statement, 40
EXPAND statement
CONSTRAINT option, 70
FIX option, 70
IIS option, 70
IMPVAR option, 70
OBJECTIVE option, 70
OMITTED option, 70
OPTMODEL procedure, 69
SOLVE option, 70
VAR option, 70

FD= option
PROC OPTMODEL statement, 40
FDIGITS= option
PROC OPTMODEL statement, 41
FEASTOL= option
PROC OPTLP statement, 578
PROC OPTMILP statement, 632
SOLVE WITH LP statement, 262
SOLVE WITH MILP statement, 328
SOLVE WITH NLP statement, 504
FILE statement
OPTMODEL procedure, 71
FINDALLSOLNS option
SOLVE WITH CLP statement, 200
FIX option
EXPAND statement, 70
FIX statement
OPTMODEL procedure, 73
FLOW= suboption
OUT= option, 389
FOR statement
OPTMODEL procedure, 73
FORCEFD= option
PROC OPTMODEL statement, 41
FORCEPRESOLVE= option
PROC OPTMODEL statement, 41
FOREST= suboption
OUT= option, 389
FORMAT=option
MPS2SASD Macro Parameters, 849
function expressions
OF keyword, 103

GCC predicate, 205
GOAL= option
TUNER statement (OPTMILP), 825
GRAPH_DIRECTION= option
SOLVE WITH NETWORK statement, 386

HEURISTICS= option

PROC OPTMILP statement, 635
SOLVE WITH MILP statement, 331
HEURISTICS= suboption
TSP= option, 396
HYBRID= option
DECOMP statement, 727

IF expression
OPTMODEL expression extensions, 106
IF statement
ELSE keyword, 74
OPTMODEL procedure, 74
THEN keyword, 74
IIS option
EXPAND statement, 70
IIS= option
PROC OPTLP statement, 577
PROC OPTQP statement, 693
SOLVE WITH LP statement, 260
SOLVE WITH NLP statement, 505
SOLVE WITH QP statement, 551
IMPVAR option
EXPAND statement, 70
IMPVAR statement
OPTMODEL procedure, 46
IN expression
OPTMODEL expression extensions, 107
IN keyword
index sets, 104
INCLUDE= suboption
LINKS= option, 387
NODES= option, 388
INCLUDE_SELFLINK option
SOLVE WITH NETWORK statement, 386
index sets
IN keyword, 104
index set expression, 107
index-set-item, 104
INIT keyword
NUMBER statement, 48
SET statement, 48
STRING statement, 48
VAR statement, 52
INITPRESOLVER= option
DECOMP_MASTER statement, 733
DECOMP_SUBPROB statement, 740
INITVAR option
PROC OPTMODEL statement, 41
INITVARS= option
DECOMP statement, 728
INTER aggregation expression
OPTMODEL expression extensions, 108
INTER expression
OPTMODEL expression extensions, 107
INTFUZZ= option
PROC OPTMODEL statement, 42
INTO keyword
READ DATA statement, 82
INTTOL= option
PROC OPTMILP statement, 632
SOLVE WITH MILP statement, 328

LEAVE statement
OPTMODEL procedure, 74
LEXICO predicate, 206
LINEAR_ASSIGNMENT option
algorithm options, 392
LINKS= option, 387
LINKS= suboption
OUT= option, 389
SUBGRAPH= option, 390
LOGFREQ= option
DECOMP statement, 728
PROC OPTLP statement, 578
PROC OPTMILP statement, 632, 725
PROC OPTQP statement, 693
SOLVE WITH LP statement, 262
SOLVE WITH MILP statement, 329
SOLVE WITH NETWORK statement, 386
SOLVE WITH NLP statement, 504
SOLVE WITH QP statement, 552
TUNER statement (OPTMILP), 825
LOGLEVEL= option
DECOMP statement, 728
PROC OPTLP statement, 579
PROC OPTMILP statement, 633
SOLVE WITH LP statement, 262
SOLVE WITH MILP statement, 329
SOLVE WITH NETWORK statement, 386
TUNER statement (OPTMILP), 825
LOWER= suboption
LINKS= option, 388
LTRIM option
READ DATA statement, 84

MASTER_IP_BEG= option
DECOMP statement, 729
MASTER_IP_END= option
DECOMP statement, 729
MASTER_IP_FREQ= option
DECOMP statement, 730
MAX aggregation expression
OPTMODEL expression extensions, 108
MAX statement
OPTMODEL procedure, 47
MAXBLOCKS= option
DECOMP statement, 730
MAXCLIQUES= suboption

CLIQUE= option, 391
MAXCOLSPASS= option
DECOMP statement, 730
MAXCONFIGS= option
TUNER statement (OPTMILP), 826
MAXCYCLES= suboption
CYCLE= option, 391
MAXITER= option
DECOMP statement, 730
PROC OPTLP statement, 579
PROC OPTQP statement, 693
SOLVE WITH LP statement, 262
SOLVE WITH NLP statement, 505
SOLVE WITH QP statement, 552
MAXLABLEN= option
PROC OPTMODEL statement, 42
MAXLEN=option
MPS2SASD Macro Parameters, 849
MAXLENGTH= suboption
CYCLE= option, 391
MAXLINKWEIGHT= suboption
CYCLE= option, 392
MAXNODES= option
PROC OPTMILP statement, 633
SOLVE WITH MILP statement, 329
MAXNODES= suboption
TSP= option, 396
MAXNODEWEIGHT= suboption
CYCLE= option, 392
MAXNUMCUTS= suboption
MINCUT= option, 393
MAXSOLNS= option
SOLVE WITH CLP statement, 200
MAXSOLS= option
PROC OPTMILP statement, 633
SOLVE WITH MILP statement, 329
MAXSOLS= suboption
TSP= option, 396
MAXTIME= option
DECOMP statement, 730
PROC OPTLP statement, 579
PROC OPTMILP statement, 633
PROC OPTQP statement, 693
SOLVE WITH CLP statement, 200
SOLVE WITH LP statement, 263
SOLVE WITH MILP statement, 329
SOLVE WITH NETWORK statement, 387
SOLVE WITH NLP statement, 505
SOLVE WITH QP statement, 552
TUNER statement (OPTMILP), 826
MAXWEIGHT= suboption
MINCUT= option, 393
METHOD= option
DECOMP statement, 730
MILP= suboption
TSP= option, 396
MIN aggregation expression
OPTMODEL expression extensions, 108
MIN statement
OPTMODEL procedure, 47
MINCOSTFLOW option
algorithm options, 393
MINCUT= option
algorithm options, 393
MINLENGTH= suboption
CYCLE= option, 392
MINLINKWEIGHT= suboption
CYCLE= option, 392
MINNODEWEIGHT= suboption
CYCLE= option, 392
MINSPANTREE= option
algorithm options, 393
MISSCHECK option
PROC OPTMODEL statement, 42
MODE= suboption
CYCLE= option, 392
MPS2SASD Macro Parameters
FORMAT=option, 849
MAXLEN=option, 849
MPSFILE=option, 849
OUTDATA=option, 849
MPSFILE=option
MPS2SASD Macro Parameters, 849
MSGLIMIT= option
PROC OPTMODEL statement, 42
MSINGULAR= option
SOLVE WITH NLP statement, 501, 514
BNDRANGE= suboption
SOLVE WITH NLP statement, 502
MS=() option
SOLVE WITH NLP statement, 502
MULTISTART=() option
SOLVE WITH NLP statement, 502
DISTTOL= suboption
SOLVE WITH NLP statement, 503
LOGLEVEL= suboption
SOLVE WITH NLP statement, 503
MAXSTARTS= suboption
SOLVE WITH NLP statement, 503
MAXTIME= suboption
SOLVE WITH NLP statement, 503
PRINTLEVEL= suboption
SOLVE WITH NLP statement, 503

NBLOCKS= option
DECOMP statement, 731
NDF= suboption
SOLVE WITH NLP statement, 502

network solver, 381
NODES= option
  SOLVE WITH NETWORK statement, 388
NODES= suboption
  OUT= option, 389
  SUBGRAPH= option, 390
NODESEL= option
  PROC OPTMILP statement, 636
  SOLVE WITH MILP statement, 332
NODESEL= suboption
  TSP= option, 397
NOINITVAR option
  PROC OPTMODEL statement, 41
NOMISSCHECK option
  PROC OPTMODEL statement, 42
NOOBJECTIVE keyword
  SOLVE statement, 89
NOPREPROCESS option
  SOLVE WITH CLP statement, 200
NOTRIM option
  READ DATA statement, 84
NTERMS= option
  SOLVE WITH NLP statement, 502
NTHREADS= option
  DECOMP statement, 731
  DECOMP_MASTER statement, 733
  DECOMP_MASTER_IP statement, 735
  DECOMP_SUBPROB statement, 740
null statement
  OPTMODEL procedure, 75
NUMBER statement
  INIT keyword, 48
  OPTMODEL procedure, 47

OBJECTIVE keyword
  SOLVE statement, 89
OBJECTIVE option
  EXPAND statement, 70
OBJLIMIT= option
  SOLVE WITH NLP statement, 505
OBJSENSE= option
  PROC OPTLP statement, 576
  PROC OPTMILP statement, 631
  PROC OPTQP statement, 693
OBJTOL= option
  SOLVE WITH CLP statement, 200
OF keyword
  function expressions, 103
OMITTED option
  EXPAND statement, 70
OPTIONMODE= option
  TUNER statement (OPTMILP), 826
OPTIONVALUES= option
  TUNER statement (OPTMILP), 827
OPTLP procedure, 574
  DECOMP_MASTER statement, 582
  DECOMP statement, 582
  DECOMP_SUBPROB statement, 582
  PERFORMANCE statement, 21, 582
OPTMILP option tuner
  PERFORMANCE statement, 824
OPTMILP procedure, 629
  DECOMP_MASTER_IP statement, 640
  DECOMP_MASTER statement, 640
  DECOMP statement, 640
  DECOMP_SUBPROB statement, 640
  PERFORMANCE statement, 21, 640
  TUNER statement, 640, 825
OPTMODEL expression extensions
  AND aggregation expression, 104
  CARD function, 105
  CROSS expression, 105
  DIFF expression, 105
  IF expression, 106
  IN expression, 107
  index set expression, 107
  INTER aggregation expression, 108
  INTER expression, 107
  MAX aggregation expression, 108
  MIN aggregation expression, 108
  OR aggregation expression, 108
  PROD aggregation expression, 109
  range expression, 109
  set constructor expression, 110
  set literal expression, 110
  SETOF aggregation expression, 111
  SLICE expression, 111
  SUM aggregation expression, 112
  SYMDIFF expression, 112
  tuple expression, 113
  UNION aggregation expression, 113
  UNION expression, 113
  WITHIN expression, 113
OPTMODEL Procedure, 36
OPTMODEL procedure
  assignment statement, 53
  CALL statement, 54
  CLOSEFILE statement, 54
  COFOR statement, 55
  CONSTRAINT statement, 44
  CONTINUE statement, 61
  CREATE DATA statement, 61
  DO statement, 66
  DO statement, iterative, 66
  DO UNTIL statement, 68
  DO WHILE statement, 68
  DROP statement, 69
  EXPAND statement, 69

FILE statement, 71
FIX statement, 73
FOR statement, 73
IF statement, 74
IMPVAR statement, 46
LEAVE statement, 74
MAX statement, 47
MIN statement, 47
null statement, 75
NUMBER statement, 47
PERFORMANCE statement, 21, 75
PRINT statement, 75
PROFILE statement, 79
PUT statement, 81
QUIT Statement, 82
READ DATA statement, 82
RESET OPTIONS statement, 86
RESTORE statement, 86
SAVE MPS statement, 86
SAVE QPS statement, 88
SET statement, 47
SOLVE statement, 89
STOP statement, 91
STRING statement, 47
SUBMIT statement, 92
UNFIX statement, 94
USE PROBLEM statement, 95
VAR statement, 52
OPTMODEL procedure, LP solver
syntax, 259
OPTMODEL procedure, MILP solver, 325
OPTMODEL procedure, NLP solver
syntax, 499
OPTMODEL procedure, QP solver
syntax, 551
OPTQP procedure, 691
PERFORMANCE statement, 21, 695
OPTTOL= option
PROC OPTLP statement, 579
PROC OPTMILP statement, 633
SOLVE WITH LP statement, 263
SOLVE WITH MILP statement, 329
SOLVE WITH NLP statement, 506
OPTVALS= option
TUNER statement (OPTMILP), 827
OR aggregation expression
OPTMODEL expression extensions, 108
ORDER= suboption
OUT= option, 390
OUT= option
SOLVE WITH NETWORK statement, 388
OUTDATA=option
MPS2SASD Macro Parameters, 849
PACK predicate, 207
PAGE keyword
PRINT statement, 76
PUT statement, 82
PARALLELMODE= option
PERFORMANCE statement, 22
PARTITIONS= suboption
OUT= option, 390
PATHS= suboption
SHORTPATH=option, 393
PDIGITS= option
PROC OPTMODEL statement, 42
PERFORMANCE statement, 21
DETAILS option, 22
OPTLP procedure, 582
OPTMILP option tuner, 824
OPTMILP procedure, 640
OPTMODEL procedure, 75
OPTQP procedure, 695
PARALLELMODE= option, 22
PMATRIX= option
PROC OPTMODEL statement, 42
PREPROCESS option
SOLVE WITH CLP statement, 200
PRESOLVER= option
PROC OPTLP statement, 578
PROC OPTMILP statement, 631
PROC OPTMODEL statement, 43
PROC OPTQP statement, 693
SOLVE WITH LP statement, 261
SOLVE WITH MILP statement, 327
SOLVE WITH QP statement, 552
PRESTOL= option
PROC OPTMODEL statement, 43
PRICETYPE= option
PROC OPTLP statement, 580
SOLVE WITH LP statement, 264
PRIMALIN option
SOLVE WITH MILP statement, 327
PRIMALIN= option
DECOMP_MASTER_IP statement, 736
DECOMP_SUBPROB statement, 740
PROC OPTLP statement, 576
PROC OPTMILP statement, 631
PRIMALOUT= option
PROC OPTLP statement, 576
PROC OPTMILP statement, 631
PROC OPTQP statement, 694
PRINT statement
OPTMODEL procedure, 75
PAGE keyword, 76
PRINTFREQ= option
PROC OPTLP statement, 578
PROC OPTMILP statement, 632, 725

PROC OPTQP statement, 693
SOLVE WITH LP statement, 262
SOLVE WITH MILP statement, 329
SOLVE WITH NLP statement, 504
SOLVE WITH QP statement, 552
PRINTLEVEL2= option
PROC OPTLP statement, 579
PROC OPTMILP statement, 633
SOLVE WITH LP statement, 262
SOLVE WITH MILP statement, 329
PRINTLEVEL= option
PROC OPTLP statement, 579
PROC OPTMILP statement, 633
PROC OPTMODEL statement, 43
PROC OPTQP statement, 694
PRIORITY= option
PROC OPTMILP statement, 636
SOLVE WITH MILP statement, 332
PROBE= option
PROC OPTMILP statement, 634
SOLVE WITH MILP statement, 330
PROBE= suboption
TSP= option, 397
PROBLEMS= option
TUNER statement (OPTMILP), 827
PROBS= option
TUNER statement (OPTMILP), 827
PROC OPTLP statement
ALGORITHM2= option, 577
ALGORITHM= option, 577
BASIS= option, 580
CROSSOVER= option, 581
DATA= option, 576
DUALIN= option, 576
DUALIZE= option, 578
DUALOUT= option, 576
FEASTOL= option, 578
IIS= option, 577
LOGFREQ= option, 578
LOGLEVEL= option, 579
MAXITER= option, 579
MAXTIME= option, 579
OBJSENSE= option, 576
OPTTOL= option, 579
PRESOLVER= option, 578
PRICETYPE= option, 580
PRIMALIN= option, 576
PRIMALOUT= option, 576
PRINTFREQ= option, 578
PRINTLEVEL2= option, 579
PRINTLEVEL= option, 579
QUEUESIZE= option, 581
SAVE_ONLY_IF_OPTIMAL option, 576
SCALE= option, 581
SEED= option, 581
SOL= option, 577
SOLVER2= option, 577
SOLVER= option, 577
STOP_DG= option, 581
STOP_DI= option, 581
STOP_PI= option, 582
TIMETYPE= option, 580
PROC OPTMILP statement
ABSOBJGAP= option, 631
CONFLICTSEARCH= option, 635
CUTCLIQUE= option, 638
CUTFLOWCOVER= option, 638
CUTFLOWPATH= option, 638
CUTGOMORY= option, 639
CUTGUB= option, 639
CUTIMPLIED= option, 639
CUTKNAPSACK= option, 639
CUTLAP= option, 639
CUTMILIFTED= option, 639
CUTMIR= option, 639
CUTMULTICOMMODITY= option, 639
CUTOFF= option, 632
CUTS= option, 639
CUTSFACTOR= option, 639
CUTSTRATEGY= option, 639
CUTZEROHALF= option, 640
DATA= option, 630
DUALOUT= option, 631
EMPHASIS= option, 632
FEASTOL= option, 632
HEURISTICS= option, 635
INTTOL= option, 632
LOGFREQ= option, 632, 725
LOGLEVEL= option, 633
MAXNODES= option, 633
MAXSOLS= option, 633
MAXTIME= option, 633
NODESEL= option, 636
OBJSENSE= option, 631
OPTTOL= option, 633
PRIMALIN= option, 631
PRIMALOUT= option, 631
PRINTFREQ= option, 632, 725
PRINTLEVEL2= option, 633
PRINTLEVEL= option, 633
PRIORITY= option, 636
PROBE= option, 634
RELOBJGAP= option, 634
RESTARTS= option, 636
SCALE= option, 634
SEED= option, 634
STRONGITER= option, 637, 725
STRONGLEN= option, 637

SYMMETRY= option, 637
TARGET= option, 634
TIMETYPE= option, 634
VARSEL= option, 637, 725
PROC OPTMODEL statement
statement options, 40
PROC OPTQP statement
DATA= option, 692
DUALOUT=option, 692
IIS= option, 693
LOGFREQ= option, 693
MAXITER= option, 693
MAXTIME= option, 693
OBJSENSE= option, 693
PRESOLVER= option, 693
PRIMALOUT= option, 694
PRINTFREQ= option, 693
PRINTLEVEL= option, 694
SAVE_ONLY_IF_OPTIMAL option, 694
STOP_DG= option, 694
STOP_DI= option, 694
STOP_PI= option, 694
TIMETYPE= option, 694
PROD aggregation expression
OPTMODEL expression extensions, 109
PUT statement
PAGE keyword, 82
PWIDTH= option
PROC OPTMODEL statement, 44

QUEUESIZE= option
PROC OPTLP statement, 581
SOLVE WITH LP statement, 264
QUIT Statement
OPTMODEL procedure, 82

range expression
OPTMODEL expression extensions, 109
READ DATA statement
COL keyword, 84
INTO keyword, 82
LTRIM option, 84
NOTRIM option, 84
OPTMODEL procedure, 82
RTRIM option, 84
TRIM option, 84
REIFY predicate, 208
RELAXINT keyword
SOLVE statement, 89
RELOBJGAP= option
DECOMP statement, 732
PROC OPTMILP statement, 634
SOLVE WITH MILP statement, 330
RELOBJGAP= suboption
TSP= option, 397
RESET OPTIONS statement
OPTMODEL procedure, 86
RESTARTS= option
PROC OPTMILP statement, 636
SOLVE WITH MILP statement, 332
RESTORE statement
OPTMODEL procedure, 86
RTRIM option
READ DATA statement, 84

SAVE MPS statement
OPTMODEL procedure, 86
SAVE QPS statement
OPTMODEL procedure, 88
SAVE_ONLY_IF_OPTIMAL option
PROC OPTLP statement, 576
PROC OPTQP statement, 694
SCALE= option
PROC OPTLP statement, 581
PROC OPTMILP statement, 634
SOLVE WITH LP statement, 264
SOLVE WITH MILP statement, 330
SEED= option
PROC OPTLP statement, 581
PROC OPTMILP statement, 634
SOLVE WITH LP statement, 265
SOLVE WITH MILP statement, 330
SOLVE WITH NLP statement, 502
set constructor expression
OPTMODEL expression extensions, 110
set literal expression
OPTMODEL expression extensions, 110
SET statement
INIT keyword, 48
OPTMODEL procedure, 47
SETOF aggregation expression
OPTMODEL expression extensions, 111
SHORTPATH= option
algorithm options, 393
SHOWPROGRESS option
SOLVE WITH CLP statement, 200
SIGSQ= option
SOLVE WITH NLP statement, 502, 515
SINK= suboption
SHORTPATH= option, 394
SLICE expression
OPTMODEL expression extensions, 111
SOL= option
DECOMP_SUBPROB statement, 739
PROC OPTLP statement, 577
SOLVE WITH LP statement, 260
SOLTYPE= option
SOLVE WITH NLP statement, 504

SOLVE option
EXPAND statement, 70
SOLVE statement
NOOBJECTIVE keyword, 89
OBJECTIVE keyword, 89
OPTMODEL procedure, 89
RELAXINT keyword, 89
VARASSIGN= option, 211
VARSELECT= option, 211
WITH keyword, 89
SOLVE WITH CLP statement
statement options, 200
SOLVE WITH LP statement
ALGORITHM2= option, 261
ALGORITHM= option, 260
BASIS= option, 264
CROSSOVER= option, 265
DECOMP_MASTER=() option, 265
DECOMP=() option, 265
DECOMP_SUBPROB=() option, 266
DUALIZE= option, 261
FEASTOL= option, 262
IIS= option, 260
LOGFREQ= option, 262
LOGLEVEL= option, 262
MAXITER= option, 262
MAXTIME= option, 263
OPTTOL= option, 263
PRESOLVER= option, 261
PRICETYPE= option, 264
PRINTFREQ= option, 262
PRINTLEVEL2= option, 262
QUEUESIZE= option, 264
SCALE= option, 264
SEED= option, 265
SOL= option, 260
SOLVER2= option, 261
SOLVER= option, 260
STOP_DG= option, 265
STOP_DI= option, 265
STOP_PI= option, 265
TIMETYPE= option, 263
SOLVE WITH MILP statement
ABSOBJGAP= option, 328
CONFLICTSEARCH= option, 332
CUTCLIQUE= option, 334
CUTFLOWCOVER= option, 335
CUTFLOWPATH= option, 335
CUTGOMORY= option, 335
CUTGUB= option, 335
CUTIMPLIED= option, 335
CUTKNAPSACK= option, 335
CUTLAP= option, 335
CUTMILIFTED= option, 335
CUTMIR= option, 335
CUTMULTICOMMODITY= option, 335
CUTOFF= option, 328
CUTS= option, 336
CUTSFACTOR= option, 336
CUTSTRATEGY= option, 336
CUTZEROHALF= option, 336
DECOMP_MASTER_IP=() option, 336
DECOMP_MASTER=() option, 336
DECOMP=() option, 336
DECOMP_SUBPROB=() option, 336
EMPHASIS= option, 328
FEASTOL= option, 328
HEURISTICS= option, 331
INTTOL= option, 328
LOGFREQ= option, 329
LOGLEVEL= option, 329
MAXNODES= option, 329
MAXSOLS= option, 329
MAXTIME= option, 329
NODESEL= option, 332
OPTTOL= option, 329
PRESOLVER= option, 327
PRIMALIN option, 327
PRINTFREQ= option, 329
PRINTLEVEL2= option, 329
PRIORITY= option, 332
PROBE= option, 330
RELOBJGAP= option, 330
RESTARTS= option, 332
SCALE= option, 330
SEED= option, 330
STRONGITER= option, 333
STRONGLEN= option, 333
SYMMETRY= option, 333
TARGET= option, 330
TIMETYPE= option, 330
VARSEL= option, 333
SOLVE WITH NETWORK statement
statement options, 386
SOLVE WITH NLP statement
ALGORITHM= option, 504
ASINGULAR= option, 500
COV= option, 500
COVEST=() option, 500
COVOUT= option, 501
COVSING= option, 501
FEASTOL= option, 504
IIS= option, 505
LOGFREQ= option, 504
MAXITER= option, 505
MAXTIME= option, 505
MSINGULAR= option, 501
BNDRANGE= suboption, 502

MS=() option, 502
MULTISTART=() option, 502
DISTTOL= suboption, 503
LOGLEVEL= suboption, 503
MAXSTARTS= suboption, 503
MAXTIME= suboption, 503
PRINTLEVEL= suboption, 503
NDF= suboption, 502
NTERMS= option, 502
OBJLIMIT= option, 505
OPTTOL= option, 506
PRINTFREQ= option, 504
SEED= option, 502
SIGSQ= option, 502
SOLTYPE= option, 504
SOLVER= option, 504
TECH= option, 504
TECHNIQUE= option, 504
TIMETYPE= option, 506
VARDEF= option, 502
SOLVE WITH QP statement
IIS= option, 551
LOGFREQ= option, 552
MAXITER= option, 552
MAXTIME= option, 552
PRESOLVER= option, 552
PRINTFREQ= option, 552
STOP_DG= option, 552
STOP_DI= option, 553
STOP_PI= option, 553
TIMETYPE= option, 553
SOLVER2= option
PROC OPTLP statement, 577
SOLVE WITH LP statement, 261
SOLVER= option
DECOMP_SUBPROB statement, 739
PROC OPTLP statement, 577
SOLVE WITH LP statement, 260
SOLVE WITH NLP statement, 504
SOURCE= suboption
SHORTPATH= option, 394
SPPATHS= suboption
OUT= option, 390
SPWEIGHTS= suboption
OUT= option, 390
STOP statement
OPTMODEL procedure, 91
STOP_DG= option
PROC OPTLP statement, 581
PROC OPTQP statement, 694
SOLVE WITH LP statement, 265
SOLVE WITH QP statement, 552
STOP_DI= option
PROC OPTLP statement, 581
PROC OPTQP statement, 694
SOLVE WITH LP statement, 265
SOLVE WITH QP statement, 553
STOP_PI= option
PROC OPTLP statement, 582
PROC OPTQP statement, 694
SOLVE WITH LP statement, 265
SOLVE WITH QP statement, 553
STRING statement
INIT keyword, 48
OPTMODEL procedure, 47
STRONGITER= option
PROC OPTMILP statement, 637, 725
SOLVE WITH MILP statement, 333
STRONGITER= suboption
TSP= option, 397
STRONGLEN= option
PROC OPTMILP statement, 637
SOLVE WITH MILP statement, 333
STRONGLEN= suboption
TSP= option, 398
SUBGRAPH= option
SOLVE WITH NETWORK statement, 390
SUBMIT statement
OPTMODEL procedure, 92
SUM aggregation expression
OPTMODEL expression extensions, 112
SYMDIFF expression
OPTMODEL expression extensions, 112
SYMMETRY= option
PROC OPTMILP statement, 637
SOLVE WITH MILP statement, 333

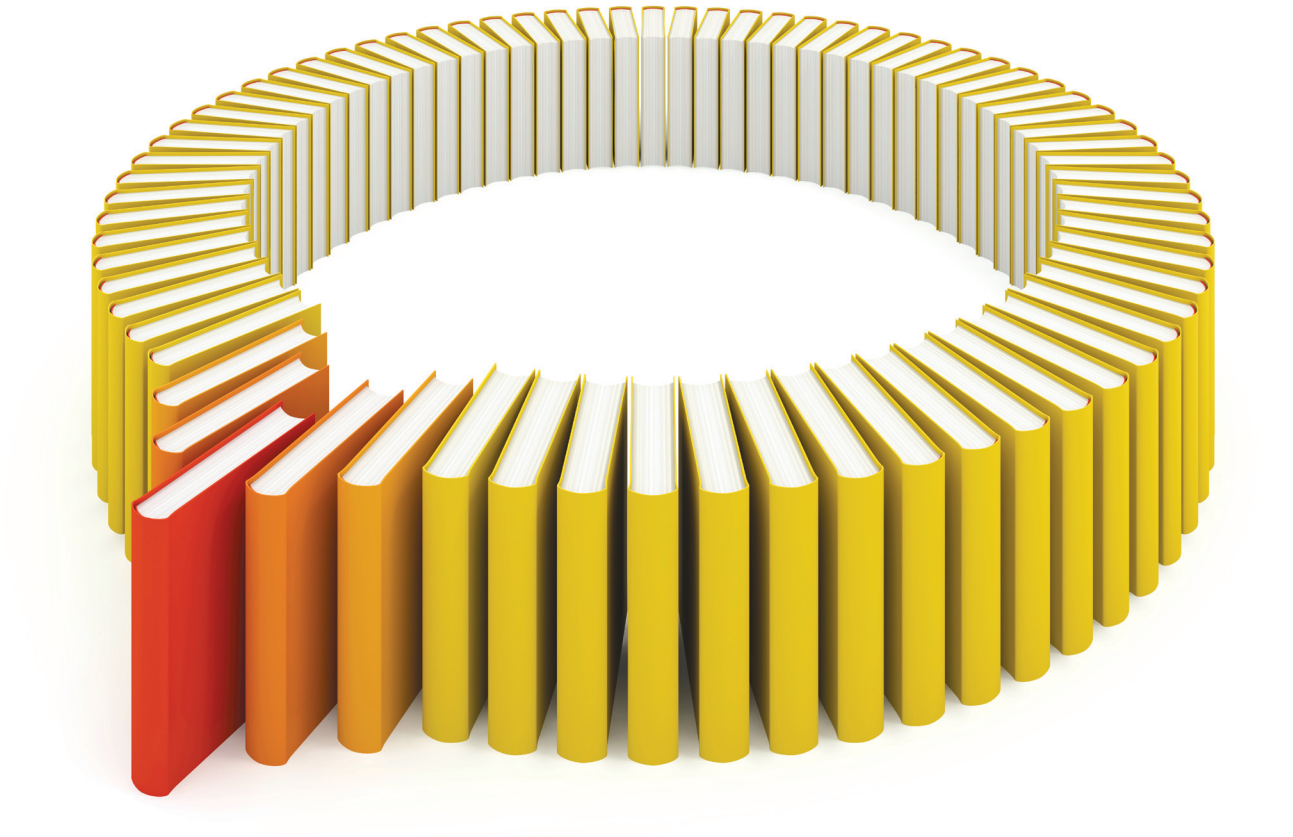
TARGET= option
PROC OPTMILP statement, 634
SOLVE WITH MILP statement, 330
TARGET= suboption
TSP= option, 398
TECH= option
SOLVE WITH NLP statement, 504
TECHNIQUE= option
SOLVE WITH NLP statement, 504
THEN keyword
IF statement, 74
TIMETYPE= option
PROC OPTLP statement, 580
PROC OPTMILP statement, 634
PROC OPTQP statement, 694
SOLVE WITH CLP statement, 200
SOLVE WITH LP statement, 263
SOLVE WITH MILP statement, 330
SOLVE WITH NETWORK statement, 387
SOLVE WITH NLP statement, 506
SOLVE WITH QP statement, 553

TOUR= suboption
    OUT= option, 390
TOUT= option
    TUNER statement (OPTMILP), 827
TRANSCL= suboption
    OUT= option, 390
TRANSITIVE_CLOSURE option
    algorithm options, 394
TRIM option
    READ DATA statement, 84
TSP option
    algorithm options, 394
TUNER statement
    OPTMILP procedure, 640
TUNER statement (OPTMILP), 825
    GOAL= option, 825
    LOGFREQ= option, 825
    LOGLEVEL= option, 825
    MAXCONFIGS= option, 826
    MAXTIME= option, 826
    OPTIONMODE= option, 826
    OPTIONVALUES= option, 827
    OPTVALS= option, 827
    PROBLEMS= option, 827
    PROBS= option, 827
    TOUT= option, 827
    TUNEROUT= option, 827
TUNEROUT= option
    TUNER statement (OPTMILP), 827
tuple expression
    OPTMODEL expression extensions, 113

UNFIX statement
    OPTMODEL procedure, 94
UNION aggregation expression
    OPTMODEL expression extensions, 113
UNION expression
    OPTMODEL expression extensions, 113
UNTIL keyword
    DO statement, iterative, 66
UPPER= suboption
    LINKS= option, 388
USE PROBLEM statement
    OPTMODEL procedure, 95
USEWEIGHT= suboption
    SHORTPATH= option, 394

VAR option
    EXPAND statement, 70
VAR statement
    INIT keyword, 52
    OPTMODEL procedure, 52
VARASSIGN= option
    SOLVE statement, 211
    SOLVE WITH CLP statement, 200
VARDEF= option
    SOLVE WITH NLP statement, 502, 513
VARFUZZ= option
    PROC OPTMODEL statement, 44
VARSEL= option
    PROC OPTMILP statement, 637, 725
    SOLVE WITH MILP statement, 333
VARSEL= suboption
    TSP= option, 398
VARSELECT= option
    SOLVE statement, 211
    SOLVE WITH CLP statement, 201

WEIGHT2= suboption
    NODES= option, 388
WEIGHT= suboption
    LINKS= option, 388
    NODES= option, 388
WHILE keyword
    DO statement, iterative, 66
WITH keyword
    SOLVE statement, 89
WITHIN expression
    OPTMODEL expression extensions, 113



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.

sas
THE POWER TO KNOW®

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies. © 2013 SAS Institute Inc. All rights reserved. S107969US.0613