Chapter 1
Introduction to Optimization

Chapter Contents

Chapter 1
Introduction to Optimization

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
- analyzing the model

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 where 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 there are several products competing 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 will 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.

A problem is formalized with the construction of a model to represent it. These models, called mathematical programs, are represented in SAS data sets and then solved using SAS/OR procedures. The solution of mathematical programs is called mathematical programming. Since mathematical programs are represented in SAS data sets, they can be saved, easily changed, and re-solved. The SAS/OR procedures also output SAS data sets containing the solutions. These can then be used to produce customized reports. In addition, this structure enables you to build decision support systems using the tools of Operations Research and other tools in the SAS System as building blocks.

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 linear programming (LP; all functions and constraints are linear) and nonlinear programming (NLP).

The data describing the model are supplied to an optimizer (such as one of the procedures described in this book), an optimizing algorithm is used to determine the optimal values for the decision variables so the objective is either maximized or minimized, the optimal values assigned to decision variables are on or between allowable bounds, and the constraints are obeyed. Determining the optimal values is the process called *optimization*.

This chapter describes how to use SAS/OR software to solve a wide variety of optimization problems. We describe various types of optimization problems, indicate which SAS/OR procedure you can use, and show how you provide data, run the procedure, and obtain optimal solutions.

In the next section we broadly classify the SAS/OR procedures based on the types of mathematical programming problems they can solve.

Linear Programming Problems

PROC OPTLP

PROC OPTLP solves linear programming problems that are submitted either in an MPS-format file or in an MPS-format SAS data set.

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, refer to Chapter 14, "The MPS-Format SAS Data Set."

PROC OPTLP provides three solvers to solve the LP: 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 more details refer to Chapter 15, "The OPTLP Procedure."

PROC OPTMODEL

PROC OPTMODEL provides a language for concisely modeling linear programming problems. The language allows a model to be expressed in a form that matches the mathematical formulation. 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 linear model in data set form for use by PROC OPTLP. For more details, refer to Chapter 6, "The OPTMODEL Procedure."

PROC LP

The LP procedure solves linear and mixed integer programs with a primal simplex solver. It can perform several types of post-optimality analysis, including range analysis, sensitivity analysis, and parametric programming. The procedure can also be used interactively.

PROC LP requires a problem data set that contains the model. In addition, a primal and active data set can be used for warm starting a problem that has been partially solved previously.

The problem data describing the model can be in one of two formats: dense or sparse. The dense format represents the model as a rectangular coefficient matrix. For details see the section "Dense Format" on page 11. The sparse format, on the other hand, represents only the nonzero elements of a rectangular coefficient matrix. See the section "Sparse Format" on page 11 for more details.

For more details on PROC LP refer to Chapter 3, "The LP Procedure."

PROC INTPOINT

The INTPOINT procedure solves linear programming problems using the interior point algorithm.

The constraint data can be specified in either the sparse or dense input format. This is the same format that is used by PROC LP; therefore, any model-building techniques that apply to models for PROC LP also apply to PROC INTPOINT.

For more details on PROC INTPOINT refer to Chapter 2, "The INTPOINT Procedure."

Network Problems

PROC NETFLOW

The NETFLOW procedure solves network flow problems with linear side constraints using either a network simplex algorithm or an interior point algorithm. In addition, it can solve linear programming (LP) problems using the interior point algorithm.

Networks and the Network Simplex Algorithm

PROC NETFLOW's network simplex algorithm solves pure network flow problems and network flow problems with linear side constraints. The procedure accepts the network specification in formats that are particularly suited to networks. Although network problems could be solved by PROC LP, the NETFLOW procedure generally solves network flow problems more efficiently than PROC LP.

Network flow problems, such as finding the minimum cost flow in a network, require model representation in a format that is specialized for network structures. The network is represented in two data sets: a node data set that names the nodes in the network and gives supply and demand information at them, and an arc data set that defines the arcs in the network using the node names and gives arc costs and capacities. In addition, a side-constraint data set is included that gives any side constraints that apply to the flow through the network. Examples of these are found later in this chapter.

The constraint data can be specified in either the sparse or dense input format. This is the same format that is used by PROC LP; therefore, any model-building techniques that apply to models for PROC LP also apply to network flow models having side constraints.

Linear and Network Programs Solved by the Interior Point Algorithm

The data required by PROC NETFLOW for a linear program resemble the data for nonarc variables and constraints for constrained network problems. They are similar to the data required by PROC LP.

The LP representation requires a data set that defines the variables in the LP using variable names, and gives objective function coefficients and upper and lower bounds. In addition, a constraint data set can be included that specifies any constraints.

When solving a constrained network problem, you can specify the INTPOINT option to indicate that the interior point algorithm is to be used. The input data are the same whether the simplex or interior point method is used. The interior point method is often faster when problems have many side constraints.

The constraint data can be specified in either the sparse or dense input format. This is the same format that is used by PROC LP; therefore, any model-building techniques that apply to models for PROC LP also apply to LP models solved by PROC NETFLOW.

PROC INTPOINT

The INTPOINT procedure solves the Network Program with Side Constraints (NPSC) problem using the interior point algorithm.

The data required by PROC INTPOINT are similar to the data required by PROC NETFLOW when solving network flow models using the interior point algorithm.

The constraint data can be specified in either the sparse or dense input format. This is the same format that is used by PROC LP and PROC NETFLOW; therefore, any model-building techniques that apply to models for PROC LP or PROC NETFLOW also apply to PROC INTPOINT.

For more details on PROC INTPOINT refer to Chapter 2, "The INTPOINT Procedure."

Mixed Integer Linear Problems

PROC OPTMILP (Experimental)

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.

The OPTMILP procedure requires a MILP to be specified using a SAS data set that adheres to the MPS format. See Chapter 14, "The MPS-Format SAS Data Set," for details about the MPS-format data set.

PROC OPTMODEL

PROC OPTMODEL provides a language for concisely modeling mixed integer linear programming problems. The language allows a model to be expressed in a form that matches the mathematical formulation. 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, refer to Chapter 6, "The OPTMODEL Procedure."

PROC LP

The LP procedure solves MILPs with a primal simplex solver. To solve a MILP you need to identify the integer variables. You can do this with a row in the input data set that has the keyword INTEGER for the type variable. It is important to note that integer variables must have upper bounds explicitly defined.

As with linear programs, you can specify MIP problem data using sparse or dense format. For more details see Chapter 3, "The LP Procedure."

Quadratic Programming Problems

PROC OPTQP

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

You can specify the problem input data in one of two formats: QPS-format flat file or QPS-format SAS data set. For details on the QPS-format data specification, refer to Chapter 14, "The MPS-Format SAS Data Set." For more details on the OPTQP procedure, refer to Chapter 17, "The OPTQP Procedure."

PROC OPTMODEL

PROC OPTMODEL provides a language for concisely modeling quadratic programming problems. The language allows a model to be expressed in a form that matches the mathematical formulation. 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, refer to Chapter 6, "The OPTMODEL Procedure."

Nonlinear Problems

PROC OPTMODEL

PROC OPTMODEL provides a language for concisely modeling nonlinear programming (NLP) problems. The language allows a model to be expressed in a form that matches the mathematical formulation. Within OPTMODEL you can declare a model, pass it directly to various solvers, and review the solver result. For more details, refer to Chapter 6, "The OPTMODEL Procedure."

You can solve the following types of nonlinear programming problems using PROC OPTMODEL:

- **Nonlinear objective function, linear constraints**: Invoke the constrained nonlinear programming (NLPC) solver. For more details about the NLPC solver, refer to Chapter 10, "The NLPC Nonlinear Optimization Solver."
- **Nonlinear objective function, nonlinear constraints**: Invoke the sequential programming (SQP) or interior point nonlinear programming (IPNLP) solver. For more details about the SQP solver, refer to Chapter 13, "The Sequential Quadratic Programming Solver." For more details about the IPNLP solver, refer to Chapter 7, "The Interior Point Nonlinear Programming Solver."
- **Nonlinear objective function, no constraints**: Invoke the unconstrained nonlinear programming (NLPU) solver. For more details about the NLPU solver, refer to Chapter 11, "The Unconstrained Nonlinear Programming Solver."

PROC NLP

The NLP procedure (**N**on**L**inear **P**rogramming) offers a set of optimization techniques for minimizing or maximizing a continuous nonlinear function subject to linear and nonlinear, equality and inequality, and lower and upper bound constraints. Problems of this type are found in many settings ranging from optimal control to maximum likelihood estimation.

Nonlinear programs can be input into the procedure in various ways. The objective, constraint, and derivative functions are specified using the programming statements of PROC NLP. In addition, information in SAS data sets can be used to define the structure of objectives and constraints, and to specify constants used in objectives, constraints, and derivatives.

PROC NLP uses the following data sets to input various pieces of information:

- The DATA= data set enables you to specify data shared by all functions involved in a least-squares problem.
- The INQUAD= data set contains the arrays appearing in a quadratic programming problem.
- The INEST= data set specifies initial values for the decision variables, the values of constants that are referred to in the program statements, and simple boundary and general linear constraints.
- The MODEL= data set specifies a model (functions, constraints, derivatives) saved at a previous execution of the NLP procedure.

As an alternative to supplying data in SAS data sets, some or all data for the model can be specified using SAS programming statements. These are similar to those used in the SAS DATA step.

For more details on PROC NLP refer to Chapter 4, “The NLP Procedure.”

Model Building

Model generation and maintenance are often difficult and expensive aspects of applying mathematical programming techniques. The flexible input formats for the optimization procedures in SAS/OR software simplify this task.

PROC LP

A small product-mix problem serves as a starting point for a discussion of different types of model formats supported in SAS/OR software.

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

Four processes are used to manufacture the candy:

1. Process 1 combines and cooks the basic ingredients for both chocolates 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 chocolates, and then cools and cuts the bars.
4. Process 4 is packaging: chocolates are 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 chocolates 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 chocolates (sec)	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: $0.25(chocolates) + 0.75(toffee)$

which is the company's total profit.

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: $15(chocolates) + 40(toffee) \leq 27{,}000$

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

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

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

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

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

This linear program illustrates the type of problem known as a product mix example. The mix of products that maximizes the objective without violating the constraints is the solution. Two formats — dense or sparse — can be used to represent this model.

Dense Format

The following DATA step creates a SAS data set for this product mix problem. Notice that the values of CHOCO and TOFFEE in the data set are the coefficients of those variables in the equations corresponding to the objective function and constraints. The variable _id_ contains a character string that names the rows in the data set. The variable _type_ is a character variable that contains keywords that describe the type of each row in the problem data set. The variable _rhs_ contains the right-hand-side values.

```
data factory;
   input _id_ $ CHOCO TOFFEE _type_ $ _rhs_;
   datalines;
object      0.25    0.75   MAX      .
process1   15.00   40.00   LE   27000
process2    0.00   56.25   LE   27000
process3   18.75    0.00   LE   27000
process4   12.00   50.00   LE   27000
;
```

To solve this problem using PROC LP, specify the following:

```
proc lp data = factory;
run;
```

Sparse Format

Typically, mathematical programming models are sparse. That is, few of the coefficients in the constraint matrix are nonzero. The dense problem format shown in the previous section is an inefficient way to represent sparse models. The LP procedure also accepts data in a sparse input format. Only the nonzero coefficients must

be specified. It is consistent with the standard MPS sparse format, and much more flexible; models using the MPS format can be easily converted to the LP format.

Although the factory example in the last section is not sparse, an example of the sparse input format for that problem is illustrated here. The sparse data set has four variables: a row type identifying variable (_type_), a row name variable (_row_), a column name variable (_col_), and a coefficient variable (_coef_).

```
data sp_factory;
   format _type_ $8. _row_ $10. _col_ $10.;
   input _type_ $_row_ $ _col_ $ _coef_ ;
   datalines;
max      object     .            .
.        object     chocolate    .25
.        object     toffee       .75
le       process1   .            .
.        process1   chocolate    15
.        process1   toffee       40
.        process1   _RHS_        27000
le       process2   .            .
.        process2   toffee       56.25
.        process2   _RHS_        27000
le       process3   .            .
.        process3   chocolate    18.75
.        process3   _RHS_        27000
le       process4   .            .
.        process4   chocolate    12
.        process4   toffee       50
.        process4   _RHS_        27000
;
```

To solve this problem using PROC LP specify the following:

```
proc lp data = sp_factory
   sparsecondata;
run;
```

The Solution Summary (shown in Figure 1.1) gives information about the solution that was found, including whether the optimizer terminated successfully after finding the optimum.

When PROC LP solves a problem, it uses an iterative process. First, the procedure finds a feasible solution that satisfies the constraints. Second, it finds the optimal solution from the set of feasible solutions. The Solution Summary lists the number of iterations in each of these phases, the number of variables in the initial feasible solution, the time the procedure required to solve the problem, and the number of matrix inversions necessary.

The LP Procedure

Solution Summary

Terminated Successfully

Objective Value	475
Phase 1 Iterations	0
Phase 2 Iterations	3
Phase 3 Iterations	0
Integer Iterations	0
Integer Solutions	0
Initial Basic Feasible Variables	6
Time Used (seconds)	0
Number of Inversions	3
Epsilon	1E-8
Infinity	1.797693E308
Maximum Phase 1 Iterations	100
Maximum Phase 2 Iterations	100
Maximum Phase 3 Iterations	99999999
Maximum Integer Iterations	100
Time Limit (seconds)	120

Figure 1.1. Solution Summary

After performing three Phase 2 iterations, the procedure terminated successfully with an optimal objective value of 475.

Separating the Data from the Model Structure

It is often desirable to keep the data separate from the structure of the model. This is useful for large models with numerous identifiable components. The data are best organized in rectangular tables that can be easily examined and modified. Then, before the problem is solved, the model is built using the stored data. This process of model building is known as *matrix generation*. In conjunction with the sparse format, the SAS DATA step provides a good matrix generation language.

For example, consider the candy manufacturing example introduced previously. Suppose that, for the user interface, it is more convenient to organize the data so that each record describes the information related to each product (namely, the contribution to the objective function and the unit amount needed for each process). A DATA step for saving the data might look like this:

```
data manfg;
   format product $12.;
   input product $ object process1 - process4 ;
   datalines;
chocolate        .25    15  0.00 18.75    12
toffee           .75    40 56.25  0.00    50
licorice        1.00    29 30.00 20.00    20
jelly_beans      .85    10  0.00 30.00    10
_RHS_            .   27000 27000 27000 27000
;
```

Notice that there is a special record at the end having product _RHS_. This record gives the amounts of time available for each of the processes. This information could have been stored in another data set. The next example illustrates a model where the data are stored in separate data sets.

Building the model involves adding the data to the structure. There are as many ways to do this as there are programmers and problems. The following DATA step shows one way to use the candy data to build a sparse format model to solve the product mix problem.

```
data model;
   array process object process1-process4;
   format _type_ $8. _row_ $12. _col_ $12. ;
   keep _type_ _row_ _col_ _coef_;

   set manfg;          /* read the manufacturing data */

   /* build the object function */

   if _n_=1 then do;
      _type_='max'; _row_='object'; _col_=' '; _coef_=.;
      output;
   end;

   /* build the constraints */

   do over process;
      if _i_>1 then do;
         _type_='le'; _row_='process'||put(_i_-1,1.);
      end;
      else             _row_='object';
      _col_=product; _coef_=process;
      output;
   end;
run;
```

The sparse format data set is shown in Figure 1.2.

Obs	_type_	_row_	_col_	_coef_
1	max	object		.
2	max	object	chocolate	0.25
3	le	process1	chocolate	15.00
4	le	process2	chocolate	0.00
5	le	process3	chocolate	18.75
6	le	process4	chocolate	12.00
7		object	toffee	0.75
8	le	process1	toffee	40.00
9	le	process2	toffee	56.25
10	le	process3	toffee	0.00
11	le	process4	toffee	50.00
12		object	licorice	1.00
13	le	process1	licorice	29.00
14	le	process2	licorice	30.00
15	le	process3	licorice	20.00
16	le	process4	licorice	20.00
17		object	jelly_beans	0.85
18	le	process1	jelly_beans	10.00
19	le	process2	jelly_beans	0.00
20	le	process3	jelly_beans	30.00
21	le	process4	jelly_beans	10.00
22		object	_RHS_	.
23	le	process1	_RHS_	27000.00
24	le	process2	_RHS_	27000.00
25	le	process3	_RHS_	27000.00
26	le	process4	_RHS_	27000.00

Figure 1.2. Sparse Data Format

The model data set looks a little different from the sparse representation of the candy model shown earlier. It not only includes additional products (licorice and jelly beans), but it also defines the model in a different order. Since the sparse format is robust, the model can be generated in ways that are convenient for the DATA step program.

If the problem had more products, you could increase the size of the manfg data set to include the new product data. Also, if the problem had more than four processes, you could add the new process variables to the manfg data set and increase the size of the process array in the model data set. With these two simple changes and additional data, a product mix problem having hundreds of processes and products can be solved.

PROC NETFLOW

Network flow problems can be described by specifying the nodes in the network and their supplies and demands, and the arcs in the network and their costs, capacities, and lower flow bounds. Consider the simple transshipment problem in Figure 1.3 as an illustration.

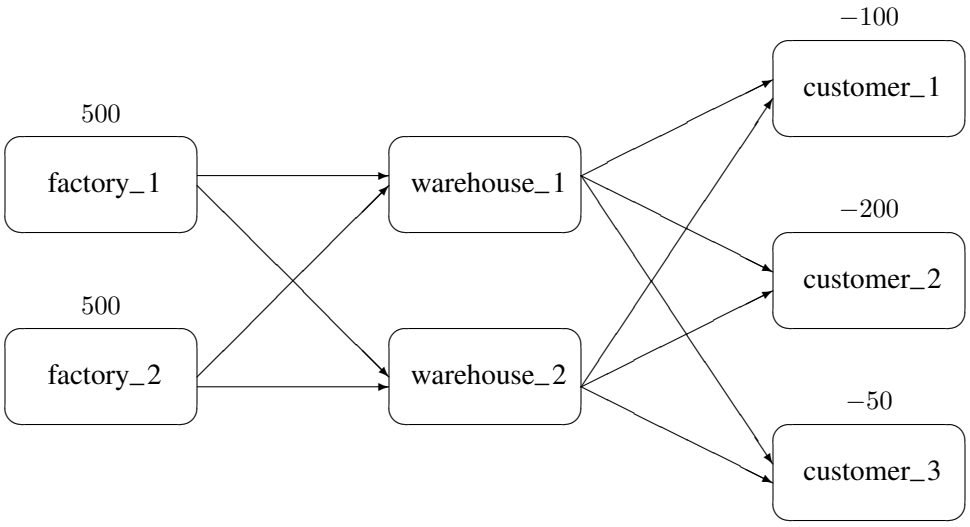

Figure 1.3. Transshipment Problem

Suppose the candy manufacturing company has two factories, two warehouses, and three customers for chocolate. The two factories each have a production capacity of 500 pounds per day. The three customers have demands of 100, 200, and 50 pounds per day, respectively.

The following data set describes the supplies (positive values for the supdem variable) and the demands (negative values for the supdem variable) for each of the customers and factories.

```
data nodes;
   format node $10. ;
   input node $ supdem;
   datalines;
customer_1   -100
customer_2   -200
customer_3    -50
factory_1     500
factory_2     500
;
```

Suppose that there are two warehouses that are used to store the chocolate before shipment to the customers, and that there are different costs for shipping between each factory, warehouse, and customer. What is the minimum cost routing for supplying the customers?

Arcs are described in another data set. Each observation defines a new arc in the network and gives data about the arc. For example, there is an arc between the node factory_1 and the node warehouse_1. Each unit of flow on that arc costs 10.

Although this example does not include it, lower and upper bounds on the flow across that arc can be listed here.

```
data network;
   format from $12. to $12.;
   input from $ to $ cost ;
   datalines;
factory_1     warehouse_1  10
factory_2     warehouse_1   5
factory_1     warehouse_2   7
factory_2     warehouse_2   9
warehouse_1   customer_1    3
warehouse_1   customer_2    4
warehouse_1   customer_3    4
warehouse_2   customer_1    5
warehouse_2   customer_2    5
warehouse_2   customer_3    6
;
```

You can use PROC NETFLOW to find the minimum cost routing. This procedure takes the model as defined in the network and nodes data sets and finds the minimum cost flow.

```
proc netflow arcout=arc_sav
             arcdata=network nodedata=nodes;
   node node;      /* node data set information */
   supdem supdem;
   tail from;      /* arc data set information */
   head to;
   cost cost;
   run;

proc print;
   var from to cost _capac_ _lo_ _supply_ _demand_
       _flow_ _fcost_ _rcost_;
   sum _fcost_;
   run;
```

PROC NETFLOW produces the following messages in the SAS log:

```
NOTE: Number of nodes= 7 .
NOTE: Number of supply nodes= 2 .
NOTE: Number of demand nodes= 3 .
NOTE: Total supply= 1000 , total demand= 350 .
NOTE: Number of arcs= 10 .
NOTE: Number of iterations performed (neglecting
      any constraints)= 7 .
NOTE: Of these, 2 were degenerate.
NOTE: Optimum (neglecting any constraints) found.
NOTE: Minimal total cost= 3050 .
NOTE: The data set WORK.ARC_SAV has 10 observations
      and 13 variables.
```

The solution (Figure 1.4) saved in the arc_sav data set shows the optimal amount of chocolate to send across each arc (the amount to ship from each factory to each warehouse and from each warehouse to each customer) in the network per day.

Obs	_from_	_to_	_cost_	_CAPAC_	_LO_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_	_RCOST_
1	warehouse_1	customer_1	3	99999999	0	.	100	100	300	.
2	warehouse_2	customer_1	5	99999999	0	.	100	0	0	4
3	warehouse_1	customer_2	4	99999999	0	.	200	200	800	.
4	warehouse_2	customer_2	5	99999999	0	.	200	0	0	3
5	warehouse_1	customer_3	4	99999999	0	.	50	50	200	.
6	warehouse_2	customer_3	6	99999999	0	.	50	0	0	4
7	factory_1	warehouse_1	10	99999999	0	500	.	0	0	5
8	factory_2	warehouse_1	5	99999999	0	500	.	350	1750	.
9	factory_1	warehouse_2	7	99999999	0	500	.	0	0	.
10	factory_2	warehouse_2	9	99999999	0	500	.	0	0	2
									====	
									3050	

Figure 1.4. ARCOUT Data Set

Notice which arcs have positive flow (_FLOW_ is greater than 0). These arcs indicate the amount of chocolate that should be sent from factory_2 to warehouse_1 and from there to the three customers. The model indicates no production at factory_1 and no use of warehouse_2.

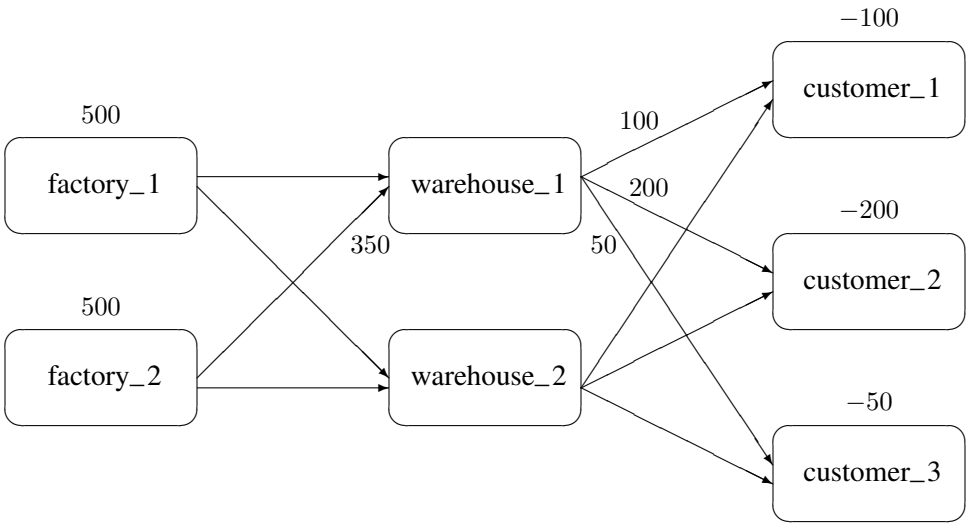

Figure 1.5. Optimal Solution for the Transshipment Problem

PROC OPTMODEL

Modeling a Linear Programming Problem

Consider the candy manufacturer's problem described in the section "PROC LP" on page 10. You can formulate the problem using PROC OPTMODEL and solve it using the primal simplex solver as follows:

```
proc optmodel;

  /* declare variables */
  var choco, toffee;

  /* 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 the following summary output:

```
                Proc OPTMODEL

               Solver Results

        Technique          Primal Simplex
        Status                    Optimal
        Iterations                      2
        Objective                  profit
        Objective Value               475


                 choco    toffee

                  1000       300
```

You can observe from the preceding example that PROC OPTMODEL provides an easy and intuitive way of modeling and solving mathematical programming models.

Modeling a Nonlinear Programming Problem

The following optimization problem illustrates how you can use some features of PROC OPTMODEL to formulate and solve nonlinear programming problems. The objective of the problem is to find coefficients for an approximation function that matches the values of a given function, $f(x)$, at a set of points P. The approximation is a rational function with degree d in the numerator and denominator:

$$r(x) = \frac{\alpha_0 + \sum_{i=1}^{d} \alpha_i x^i}{\beta_0 + \sum_{i=1}^{d} \beta_i x^i}$$

The problem can be formulated by minimizing the sum of squared errors at each point in P:

$$\min \quad \sum_{x \in P} [r(x) - f(x)]^2$$

The following code implements this model. The function $f(x) = 2^x$ is approximated over a set of points P in the range 0 to 1. The function values are saved in a data set that is used by PROC OPTMODEL to set model parameters:

```
data points;
   /* generate data points */
   keep f x;
   do i = 0 to 100;
      x = i/100;
      f = 2**x;
      output;
   end;

proc optmodel;
   /* declare, read, and save our data points */
   set points;
   number f{points};
   read data points into points = [x] f;

   /* declare variables and model parameters */
   number d=1; /* linear polynomial */
   var a{0..d};
   var b{0..d} init 1;
   constraint fixb0: b[0] = 1;

   /* minimize sum of squared errors */
   min z=sum{x in points}
               ((a[0] + sum{i in 1..d} a[i]*x**i) /
                (b[0] + sum{i in 1..d} b[i]*x**i) - f[x])**2;

   /* solve and show coefficients */
   solve;
   print a b;
   quit;
```

The expression for the objective z is defined using operators that parallel the mathematical form. In this case the polynomials in the rational function are linear, so d is equal to 1.

The constraint fixb0 forces the constant term of the rational function denominator, b[0], to equal 1. This causes the resulting coefficients to be normalized. The OPTMODEL presolver preprocesses the problem to remove the constraint. An unconstrained solver is used after substituting for b[0].

The SOLVE statement selects a solver, calls it, and displays the status. The PRINT command then prints the values of coefficient arrays a and b:

```
Proc OPTMODEL

Solver Results

Technique                    L-BFGS
Status                       Normal
Iterations                       10
Objective                         z
Objective Value          0.0000591


 [1]              a              b

  0         0.99817        1.00000
  1         0.42064       -0.29129
```

The approximation for $f(x) = 2^x$ between 0 and 1 is therefore

$$f_{\text{approx}}(x) = \frac{0.99817 + 0.42064x}{1 - 0.29129x}$$

Matrix Generation

It is desirable to keep data in separate tables, and then to automate model building and reporting. This example illustrates a problem that has elements of both a product mix problem and a blending problem. Suppose four kinds of ties are made: all silk, all polyester, a 50-50 polyester-cotton blend, and a 70-30 cotton-polyester blend.

The data include cost and supplies of raw material, selling price, minimum contract sales, maximum demand of the finished products, and the proportions of raw materials that go into each product. The objective is to find the product mix that maximizes profit.

The data are saved in three SAS data sets. The program that follows demonstrates one way for these data to be saved.

```
data material;
   format descpt $20.;
   input descpt $ cost supply;
   datalines;
silk_material             .21   25.8
polyester_material        .6    22.0
cotton_material           .9    13.6
;

data tie;
   format descpt $20.;
   input descpt $ price contract demand;
   datalines;
all_silk                 6.70    6.0    7.00
all_polyester            3.55   10.0   14.00
poly_cotton_blend        4.31   13.0   16.00
cotton_poly_blend        4.81    6.0    8.50
;

data manfg;
   format descpt $20.;
   input descpt $ silk poly cotton;
   datalines;
all_silk                  100     0      0
all_polyester               0   100      0
poly_cotton_blend           0    50     50
cotton_poly_blend           0    30     70
;
```

The following program takes the raw data from the three data sets and builds a linear program model in the data set called model. Although it is designed for the three-resource, four-product problem described here, it can easily be extended to include more resources and products. The model-building DATA step remains essentially the same; all that changes are the dimensions of loops and arrays. Of course, the data tables must expand to accommodate the new data.

```
data model;
   array  raw_mat {3} $ 20 ;
   array  raw_comp {3} silk poly cotton;
   length _type_ $ 8  _col_ $ 20  _row_ $ 20 _coef_ 8 ;
   keep   _type_      _col_       _row_      _coef_ ;

   /* define the objective, lower, and upper bound rows */

   _row_='profit'; _type_='max';     output;
   _row_='lower';  _type_='lowerbd'; output;
   _row_='upper';  _type_='upperbd'; output;
   _type_=' ';

   /* the object and upper rows for the raw materials */

   do i=1 to 3;
      set material;
      raw_mat[i]=descpt;  _col_=descpt;
      _row_='profit';    _coef_=-cost;  output;
      _row_='upper';     _coef_=supply; output;
   end;

   /* the object, upper, and lower rows for the products */

   do i=1 to 4;
      set tie;
      _col_=descpt;
      _row_='profit'; _coef_=price;    output;
      _row_='lower';  _coef_=contract; output;
      _row_='upper';  _coef_=demand;   output;
   end;

   /* the coefficient matrix for manufacturing */

   _type_='eq';
   do i=1 to 4;          /* loop for each raw material */
      set manfg;
      do j=1 to 3;       /* loop for each product      */

         _col_=descpt;   /* % of material in product   */
         _row_  = raw_mat[j];
         _coef_ = raw_comp[j]/100;
         output;

         _col_  = raw_mat[j];  _coef_ = -1;
         output;

         /* the right-hand side */

         if i=1 then do;
            _col_='_RHS_';
            _coef_=0;
            output;
         end;
```

```
      end;
      _type_='   ';
   end;
   stop;
run;
```

The model is solved using PROC LP, which saves the solution in the PRIMALOUT data set named solution. PROC PRINT displays the solution, shown in Figure 1.6.

```
proc lp sparsedata primalout=solution;

proc print ;
   id _var_;
   var _lbound_--_r_cost_;
run;
```

VAR	_LBOUND_	_VALUE_	_UBOUND_	_PRICE_	_R_COST_
all_polyester	10	11.800	14.0	3.55	0.000
all_silk	6	7.000	7.0	6.70	6.490
cotton_material	0	13.600	13.6	-0.90	4.170
cotton_poly_blend	6	8.500	8.5	4.81	0.196
polyester_material	0	22.000	22.0	-0.60	2.950
poly_cotton_blend	13	15.300	16.0	4.31	0.000
silk_material	0	7.000	25.8	-0.21	0.000
PHASE_1_OBJECTIVE	0	0.000	0.0	0.00	0.000
profit	0	168.708	1.7977E308	0.00	0.000

Figure 1.6. Solution Data Set

The solution shows that 11.8 units of polyester ties, 7 units of silk ties, 8.5 units of the cotton-polyester blend, and 15.3 units of the polyester-cotton blend should be produced. It also shows the amounts of raw materials that go into this product mix to generate a total profit of 168.708.

Exploiting Model Structure

Another example helps to illustrate how the model can be simplified by exploiting the structure in the model when using the NETFLOW procedure.

Recall the chocolate transshipment problem discussed previously. The solution required no production at factory_1 and no storage at warehouse_2. Suppose this solution, although optimal, is unacceptable. An additional constraint requiring the production at the two factories to be balanced is needed. Now, the production at the two factories can differ by, at most, 100 units. Such a constraint might look like this:

```
-100 <= (factory_1_warehouse_1 + factory_1_warehouse_2 -
         factory_2_warehouse_1 - factory_2_warehouse_2) <= 100
```

The network and supply and demand information are saved in the following two data sets:

```
data network;
   format from $12. to $12.;
   input  from $ to $ cost ;
   datalines;
factory_1    warehouse_1  10
factory_2    warehouse_1   5
factory_1    warehouse_2   7
factory_2    warehouse_2   9
warehouse_1  customer_1   3
warehouse_1  customer_2   4
warehouse_1  customer_3   4
warehouse_2  customer_1   5
warehouse_2  customer_2   5
warehouse_2  customer_3   6
;

data nodes;
   format node $12. ;
   input node $  supdem;
   datalines;
customer_1   -100
customer_2   -200
customer_3    -50
factory_1     500
factory_2     500
;
```

The factory-balancing constraint is not a part of the network. It is represented in the sparse format in a data set for side constraints.

```
data side_con;
   format _type_ $8. _row_ $8. _col_ $21. ;
   input  _type_  _row_  _col_  _coef_ ;
   datalines;
eq       balance   .                        .
.        balance  factory_1_warehouse_1     1
.        balance  factory_1_warehouse_2     1
.        balance  factory_2_warehouse_1    -1
.        balance  factory_2_warehouse_2    -1
.        balance  diff                     -1
lo       lowerbd  diff                   -100
up       upperbd  diff                    100
;
```

This data set contains an equality constraint that sets the value of DIFF to be the amount that factory 1 production exceeds factory 2 production. It also contains implicit bounds on the DIFF variable. Note that the DIFF variable is a nonarc variable.

You can use the following call to PROC NETFLOW to solve the problem:

```
proc netflow
   conout=con_sav
```

```
      arcdata=network nodedata=nodes condata=side_con
      sparsecondata ;
      node node;
      supdem supdem;
      tail from;
      head to;
      cost cost;
      run;

   proc print;
      var from to _name_ cost _capac_ _lo_ _supply_ _demand_
          _flow_ _fcost_ _rcost_;
      sum _fcost_;
      run;
```

The solution is saved in the con_sav data set, as displayed in Figure 1.7.

Obs	from	to	_NAME_	cost	_CAPAC_	_LO_	_SUPPLY_	_DEMAND_	_FLOW_	_FCOST_	_RCOST_
1	warehouse_1	customer_1		3	99999999	0	.	100	100	300	.
2	warehouse_2	customer_1		5	99999999	0	.	100	0	0	1.0
3	warehouse_1	customer_2		4	99999999	0	.	200	75	300	.
4	warehouse_2	customer_2		5	99999999	0	.	200	125	625	.
5	warehouse_1	customer_3		4	99999999	0	.	50	50	200	.
6	warehouse_2	customer_3		6	99999999	0	.	50	0	0	1.0
7	factory_1	warehouse_1		10	99999999	0	500	.	0	0	2.0
8	factory_2	warehouse_1		5	99999999	0	500	.	225	1125	.
9	factory_1	warehouse_2		7	99999999	0	500	.	125	875	.
10	factory_2	warehouse_2		9	99999999	0	500	.	0	0	5.0
11			diff	0	100	-100	.	.	-100	0	1.5
										3425	

Figure 1.7. CON_SAV Data Set

Notice that the solution now has production balanced across the factories; the production at factory 2 exceeds that at factory 1 by 100 units.

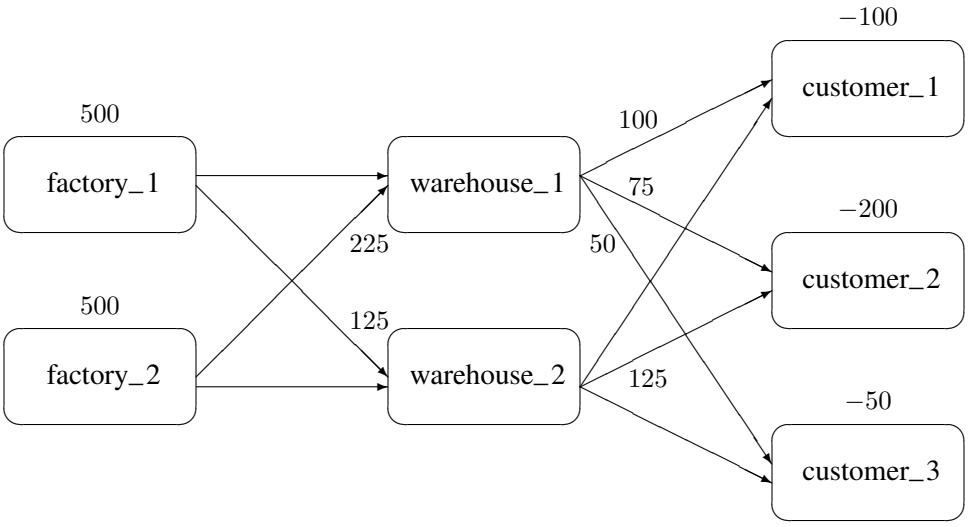

Figure 1.8. Constrained Optimum for the Transshipment Problem

Report Writing

The reporting of the solution is also an important aspect of modeling. Since the optimization procedures save the solution in one or more SAS data sets, reports can be written using any of the tools in the SAS language.

The DATA Step

Use of the DATA step and PROC PRINT is the most common way to produce reports. For example, from the data set solution shown in Figure 1.6, a table showing the revenue of the optimal production plan and a table of the cost of material can be produced with the following program.

```
data product(keep= _var_ _value_ _price_ revenue)
     material(keep=_var_ _value_ _price_ cost);
   set solution;
   if _price_>0 then do;
      revenue=_price_*_value_; output product;
   end;
   else if _price_<0 then do;
      _price_=-_price_;
      cost = _price_*_value_; output material;
   end;
run;


/* display the product report */

proc print data=product;
   id _var_;
   var _value_ _price_ revenue ;
   sum revenue;
   title 'Revenue Generated from Tie Sales';
run;

/* display the materials report */

proc print data=material;
   id _var_;
   var _value_ _price_ cost;
   sum cost;
   title 'Cost of Raw Materials';
run;
```

This DATA step reads the solution data set saved by PROC LP and segregates the records based on whether they correspond to materials or products—namely whether the contribution to profit is positive or negative. Each of these is then displayed to produce Figure 1.9.

Revenue Generated from Tie Sales

VAR	_VALUE_	_PRICE_	revenue
all_polyester	11.8	3.55	41.890
all_silk	7.0	6.70	46.900
cotton_poly_blend	8.5	4.81	40.885
poly_cotton_blend	15.3	4.31	65.943
			=======
			195.618

Cost of Raw Materials

VAR	_VALUE_	_PRICE_	cost
cotton_material	13.6	0.90	12.24
polyester_material	22.0	0.60	13.20
silk_material	7.0	0.21	1.47
			=====
			26.91

Figure 1.9. Tie Problem: Revenues and Costs

Other Reporting Procedures

The GCHART procedure can be a useful tool for displaying the solution to mathematical programming models. The con_solv data set that contains the solution to the balanced transshipment problem can be effectively displayed using PROC GCHART. In Figure 1.10, the amount that is shipped from each factory and warehouse can be seen by submitting the following SAS code:

```
title;
proc gchart data=con_sav;
   hbar from / sumvar=_flow_;
run;
```

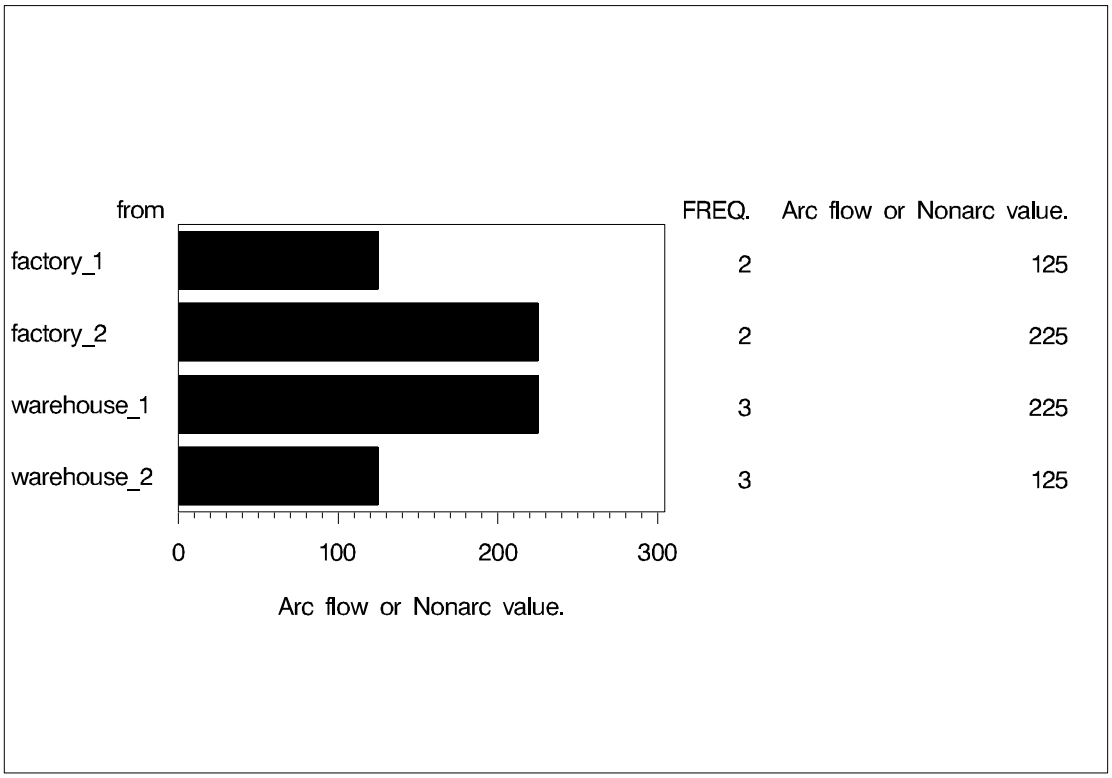

Figure 1.10. Tie Problem: Throughputs

The horizontal bar chart is just one way of displaying the solution to a mathematical program. The solution to the Tie Product Mix problem that was solved using PROC LP can also be illustrated using PROC GCHART. Here, a pie chart shows the relative contribution of each product to total revenues.

```
proc gchart data=product;
   pie _var_ / sumvar=revenue;
title 'Projected Tie Sales Revenue';
run;
```

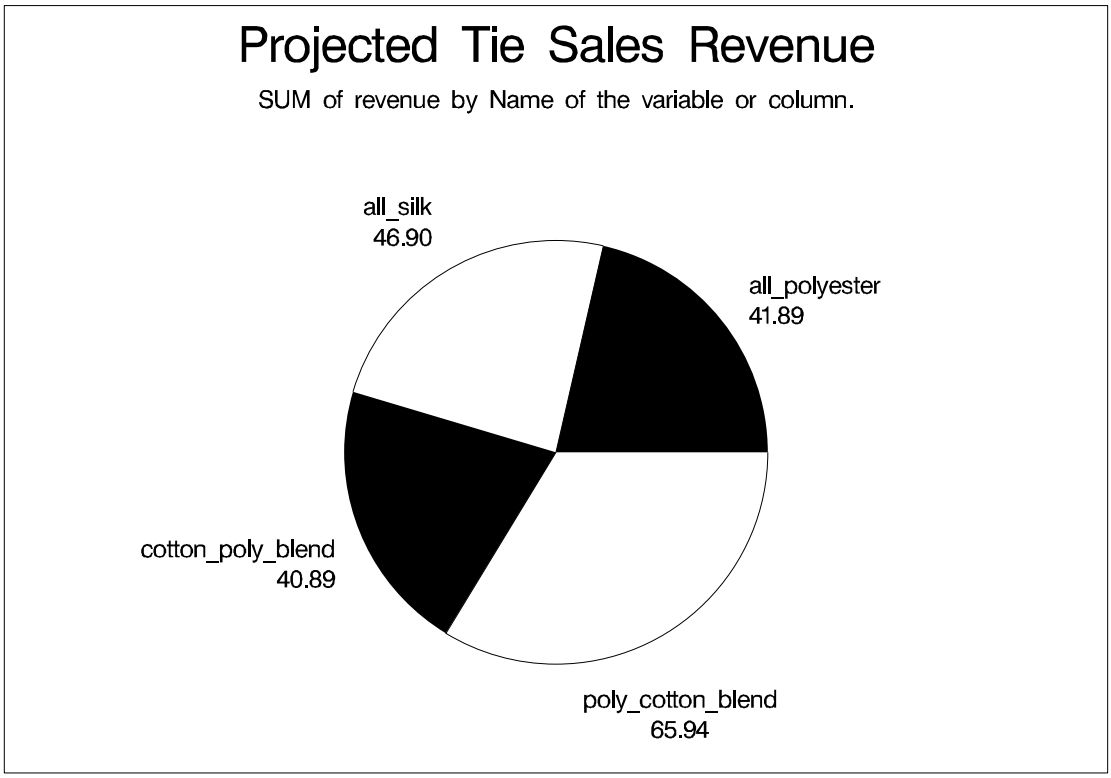

Figure 1.11. Tie Problem: Projected Tie Sales Revenue

The TABULATE procedure is another procedure that can help automate solution reporting. Several examples in Chapter 3, "The LP Procedure," illustrate its use.

References

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

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

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