SAS/OR® 13.2 User's Guide: Mathematical Programming

The Mixed Integer Linear Programming Solver

Chapter 8
The Mixed Integer Linear Programming Solver

Contents

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

$$\begin{array}{rll} \min & \mathbf{c}^T\mathbf{x} & \\ \text{subject to} & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} & \text{(MILP)} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} & \\ & \mathbf{x}_i \in \mathbb{Z} \quad \forall i \in \mathcal{S} & \end{array}$$

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 an 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:

$$
\begin{array}{llllllll}
\min & 2x_1 & - & 3x_2 & - & 4x_3 & & & \\
\text{s.t.} & & - & 2x_2 & - & 3x_3 & \geq & -5 & \text{(R1)} \\
 & x_1 & + & x_2 & + & 2x_3 & \leq & 4 & \text{(R2)} \\
 & x_1 & + & 2x_2 & + & 3x_3 & \leq & 7 & \text{(R3)} \\
 & & & x_1, & x_2, & x_3 & \geq & 0 & \\
 & & & x_1, & x_2, & x_3 & \in \mathbb{Z} & &
\end{array}
$$

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=
Emphasizes feasibility or optimality	EMPHASIS=

Table 8.1 (continued)

Description	Option
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 whether to enable or disable parallel processing of the branch-and-cut algorithm	PARALLEL=
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 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 with 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 used 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 (and including) 1E–4 and 1E–9. The default value is 1E–6.

If the MILP solver fails to find a feasible solution within this tolerance but does find a solution with a slightly larger violation, then the solver ends 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 0.0 and 0.5. The MILP solver attempts to find an optimal solution whose integer infeasibility is less than *number*. If you assign a value smaller than 1E–10 to *number* and the best solution found by the solver has integer infeasibility between *number* and 1E–10, then the solver terminates with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMODEL_ " on page 344). The default value is 1E–5.

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.

PARALLEL=*number* **|** *string*

indicates whether to enable parallel processing of the branch-and-cut algorithm. Table 8.5 describes the valid values of the PARALLEL= option.

Table 8.5 Values for PARALLEL= Option

number	*string*	**Description**
0	OFF	Disables parallel processing of the branch-and-cut algorithm
1	ON	Enables parallel processing of the branch-and-cut algorithm

The default value is 0. You can specify options for controlling parallel processing in the PERFORMANCE statement, which is documented in the section "PERFORMANCE Statement" on page 23 in Chapter 4, "Shared Concepts and Topics." The PARALLEL= option is ignored when the solver is invoked inside a COFOR loop of the OPTMODEL procedure.

PROBE=*number* **|** *string*

specifies a probing *string* or its corresponding value *number*, as listed in Table 8.6.

Table 8.6 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

$$\mid \text{BestInteger} - \text{BestBound} \mid / \, (1\text{E}{-}10 + \mid \text{BestBound} \mid)$$

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 8.7 describes the valid values of the TIMETYPE= option.

Table 8.7 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 8.8 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 23 in Chapter 4, “Shared Concepts and Topics.”

Table 8.8 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 PARALLEL=ON	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 8.9.

Table 8.9 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 8.10.

Table 8.10 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 8.11.

Table 8.11 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 340 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 8.12.

Table 8.12 Values for RESTARTS= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses a restarting strategy determined by the MILP solver
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 8.13.

Table 8.13 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 8.14.

Table 8.14 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 339.

Cut Options

Table 8.15 describes the *string* and *number* values for the cut options in the OPTMODEL procedure.

Table 8.15 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, . . . , CUTMIR=–1, and CUTZEROHALF=–1. Table 8.15 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=, 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 8.15 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 8.15 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 8.15 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 8.15 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 8.15 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 8.15 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 8.15 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 8.15 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 8.15 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 8.15 lists the values that can be assigned to *option* and *number*. The CUTMIR= 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 8.15 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

$$\begin{array}{rl} \min & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x}\ \{\geq, =, \leq\}\ \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{array}$$

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 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

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and}$$
$$\text{INTTOL} \le f_i(k) \le 1 - \text{INTTOL}\}$$

MININFEAS chooses as the branching variable the variable x_i such that i minimizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in \mathcal{S} \text{ and}$$
$$\text{INTTOL} \le f_i(k) \le 1 - \text{INTTOL}\}$$

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 136 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 337, 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 8.16 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 8.16 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

You can set levels for individual cuts by using the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIMPLIED=, CUTKNAPSACK=, CUTLAP=, CUTMILIFTED=, CUTMIR=, and CUTZEROHALF= options. The valid levels for these options are listed in Table 8.15.

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 8.9.

Parallel Processing

The branch-and-cut algorithm can be run in single-machine mode (in single-machine mode, the computation is executed by multiple threads on a single computer). To enable parallel processing of the branch-and-cut algorithm, you need to specify PARALLEL=1 in the MILP solver invocation.

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 23 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 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 10 variables, 2 constraints, and 20 constraint
      coefficients.
NOTE: The MILP solver is called.
          Node  Active   Sols    BestInteger      BestBound      Gap    Time
             0       1      3     85.0000000    178.0000000   52.25%       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     88.0955497    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      4     85.0000000     87.8539763    3.25%       0
             0       1      4     85.0000000     87.7208690    3.10%       0
             0       1      4     85.0000000     87.7180302    3.10%       0
             0       1      4     85.0000000     87.7133502    3.09%       0
             0       1      4     85.0000000     87.7128245    3.09%       0
             0       1      4     85.0000000     87.7124806    3.09%       0
NOTE: The MILP solver added 2 cuts with 8 cut coefficients at the root.
             5       3      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	2.000000
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	2.000000

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

$$\mid \text{BestInteger} - \text{BestBound} \mid / \ (1\text{E}{-}10 + \mid \text{BestBound} \mid)$$

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 $\mid \text{BestInteger} - \text{BestBound} \mid$.

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 5, "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:

$$
\begin{array}{llcll}
\max & \displaystyle\sum_{ijk} p_{ijk} x_{ijk} & & & \\
\text{s.t.} & \displaystyle\sum_{i} x_{ijk} & = & 1 \quad \forall j, k & \text{(OneEmpPerSlot)} \\
& x_{i4k} + x_{i5k} & \le & 1 \quad \forall i, k & \text{(EmpMustHaveLunch)} \\
& x_{i,\ell,k} + x_{i,\ell+1,k} + x_{i,\ell+2,k} & \le & 2 \quad \forall i, k, \text{and } l \le 6 & \text{(AtMost2ConSlots)} \\
& \displaystyle\sum_{jk} x_{ijk} & \le & h_i \quad \forall i & \text{(WeeklyHoursLimit)} \\
& x_{ijk} & = & 0 \quad \forall i, j, k \text{ s.t. } p_{ijk} > 0 & \\
& x_{ijk} \in \{0, 1\} & & \quad\ \ \forall i, j, k &
\end{array}
$$

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

		mon	tue	wed	thu	fri
slot	**name**					
1	**marc**	xxx	xxx	xxx	xxx	xxx
2	**marc**		xxx	xxx	xxx	xxx
	mike	xxx				
3	**bill**				xxx	xxx
	mike	xxx	xxx	xxx		
4	**bob**	xxx	xxx	xxx		
	mike				xxx	xxx
5	**bill**	xxx	xxx	xxx		xxx
	bob				xxx	
6	**bob**			xxx		xxx
	mike	xxx	xxx		xxx	
7	**bob**	xxx		xxx	xxx	xxx
	mike		xxx			
8	**bob**	xxx	xxx		xxx	
	mike			xxx		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 5, “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

$$x_{ijc} \le u_{ijc} = \sum_{k \in N | d_{kc} < 0} (-d_{kc})$$

This model can be represented using the following mixed integer linear program:

$$
\begin{array}{llllll}
\min & \displaystyle\sum_{(i,j)\in A}\sum_{c\in C} s_{ijc}x_{ijc} + \sum_{(i,j)\in A} f_{ij}y_{ij} & & & & \\
\text{s.t.} & \displaystyle\sum_{j\in N|(i,j)\in A} x_{ijc} - \sum_{j\in N|(j,i)\in A} x_{jic} & \le & d_{ic} & \forall i \in N, c\in C & \text{(balance\_con)} \\
 & x_{ijc} & \le & u_{ijc}y_{ij} & \forall (i,j)\in A, c\in C & \text{(fixed\_charge\_con)} \\
 & x_{ijc} & \ge & 0 & \forall (i,j)\in A, c\in C & \\
 & y_{ij} \in \{0,1\} & & & \forall (i,j)\in A &
\end{array}
$$

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

Output 8.2.1 *continued*

[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

	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	1

Output 8.2.1 *continued*

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	TotalCost
Solution Status	Optimal within Relative Gap
Objective Value	42825
Relative Gap	2.3350852E-7
Absolute Gap	0.01
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	1.110223E-16
Best Bound	42824.99
Nodes	1
Iterations	38
Presolve Time	0.00
Solution Time	0.02

[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	10
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	90
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:

$$
\begin{array}{llllll}
\min & \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_j y_j & & & & \\
\text{s.t.} & \sum_{j \in F} x_{ij} & = & 1 & \forall i \in L & \text{(assign\_def)} \\
& x_{ij} & \leq & y_j & \forall i \in L, j \in F & \text{(link)} \\
& \sum_{i \in L} d_i x_{ij} & \leq & C y_j & \forall j \in F & \text{(capacity)} \\
& x_{ij} \in \{0,1\} & & & \forall i \in L, j \in F & \\
& y_j \in \{0,1\} & & & \forall j \in F &
\end{array}
$$

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.
          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       3    966.4832160    961.2403449    0.55%       0
             0       1       3    966.4832160    962.9120844    0.37%       0
             0       1       3    966.4832160    962.9120844    0.37%       0
             0       1       3    966.4832160    962.9120844    0.37%       0
NOTE: The MILP presolver is applied again.
             0       1       4    966.4832160    962.9120844    0.37%       0
             0       1       5    966.4832160    966.4832160    0.00%       0
             0       0       5    966.4832160    966.4832160    0.00%       0
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

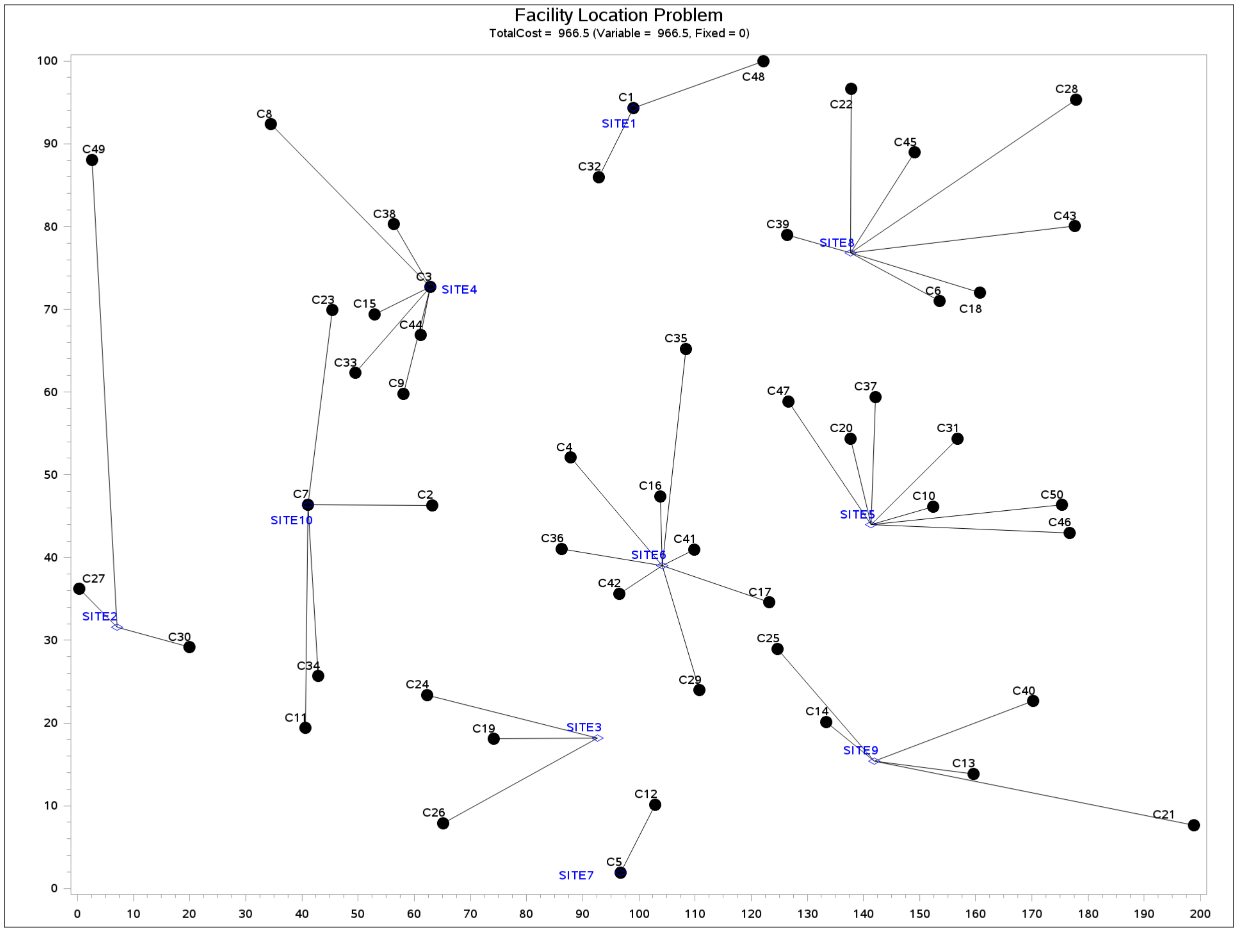
```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 510 variables (0 free, 0 fixed).
NOTE: The problem uses 1 implicit variables.
NOTE: The problem has 510 binary and 0 integer variables.
NOTE: The problem has 560 linear constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem 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.
          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  16034.0651055   9981.2701907   60.64%       0
             0       1       4  16034.0651055   9981.2701907   60.64%       0
NOTE: The MILP solver added 20 cuts with 631 cut coefficients at the root.
           264     125       7  11365.1547459  10527.4665245    7.96%       0
           287      10       9  10960.8997578  10943.8749703    0.16%       0
           295      14      10  10959.4361909  10944.1167370    0.14%       0
           299       6      11  10950.3308631  10945.0380192    0.05%       0
           315       7      12  10950.0345545  10946.4662518    0.03%       0
           323      14      13  10950.0345545  10946.4662518    0.03%       0
           325       6      14  10948.4603465  10946.4951518    0.02%       0
           332       6      14  10948.4603465  10947.3824040    0.01%       0
NOTE: Optimal within relative gap.
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.

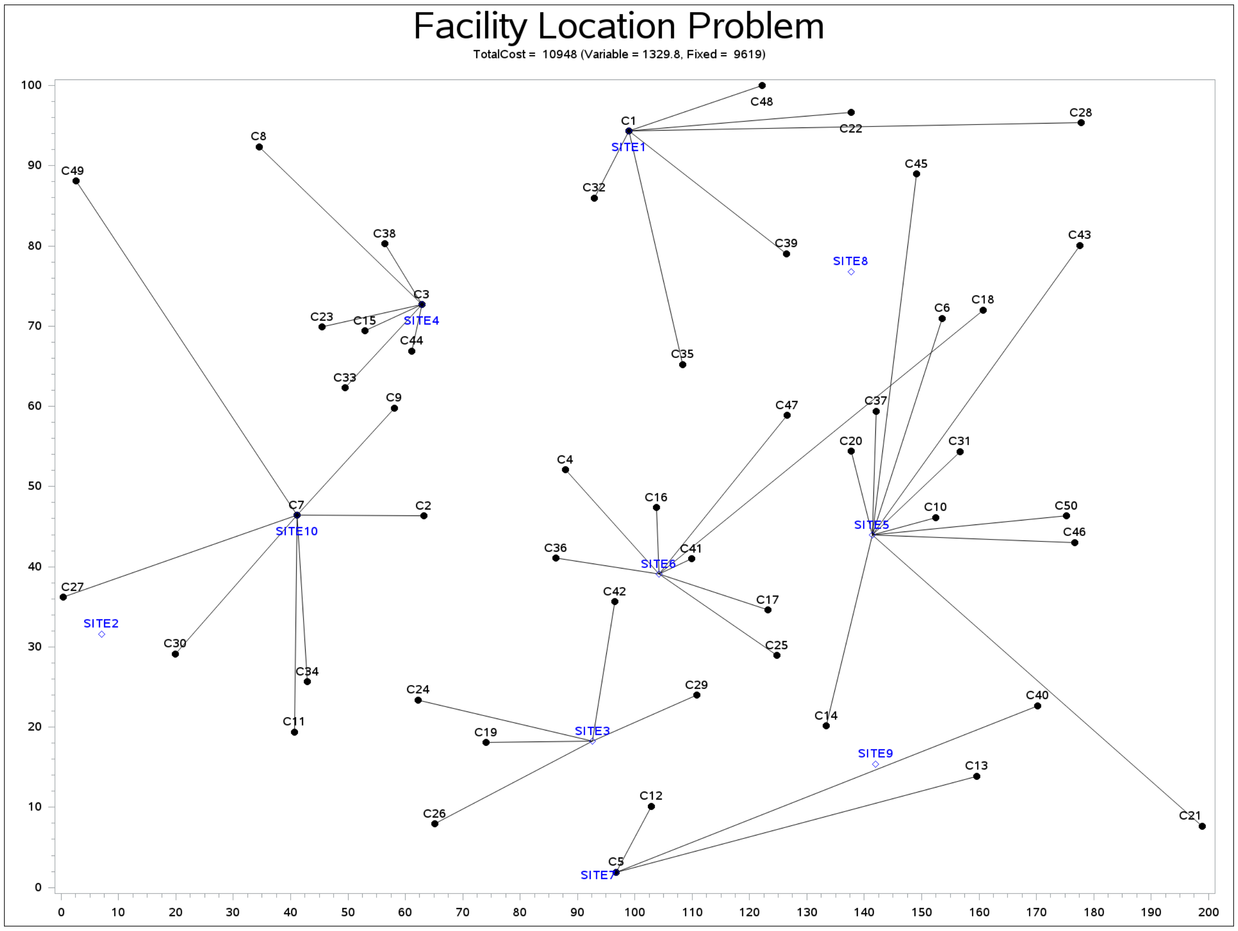
Output 8.3.3 Solution Plot for Facility Location with No Fixed Charges



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.
          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   1788.0570704    6.62%       0
             0       1       5   1906.4633474   1788.1857838    6.61%       0
             0       1       5   1906.4633474   1793.1018884    6.32%       0
             0       1       5   1906.4633474   1794.0969506    6.26%       0
             0       1       5   1906.4633474   1794.9102303    6.21%       0
             0       1       5   1906.4633474   1795.0361850    6.21%       0
             0       1       5   1906.4633474   1795.0773286    6.21%       0
             0       1       5   1906.4633474   1795.0825886    6.20%       0
             0       1       5   1906.4633474   1795.0869957    6.20%       0
             0       1       5   1906.4633474   1795.0869957    6.20%       0
             0       1       5   1906.4633474   1795.0871903    6.20%       0
NOTE: The MILP solver added 30 cuts with 668 cut coefficients at the root.
           248     228       6   1838.5150486   1800.4972361    2.11%       0
          1000     218       6   1838.5150486   1804.4864009    1.89%       1
          1100     271       7   1833.3994654   1804.4864009    1.60%       1
          1135     294       8   1833.2698616   1804.4864009    1.60%       1
          1712     590       9   1829.0025678   1811.9146769    0.94%       1
          1818     493      10   1825.1666003   1812.5246843    0.70%       1
          1967     395      12   1823.3288813   1814.9655663    0.46%       1
          1995     198      13   1819.9124343   1815.2227512    0.26%       1
          2000     200      13   1819.9124343   1815.2252293    0.26%       1
          2245       6      13   1819.9124343   1819.7532289    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:

$$\begin{array}{llllll}
\min & \sum_{e \in E} c_e x_e & & & & \\
\text{s.t.} & \sum_{e \in \delta(i)} x_e & = & 2 & \forall i \in V & \text{(two\_match)} \\
 & \sum_{e \in \delta(S)} x_e & \geq & 2 & \forall S \subset V,\ 2 \leq |S| \leq |V|-1 & \text{(subtour\_elim)} \\
 & x_e \in \{0,1\} & & & \forall e \in E &
\end{array}$$

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):

$$\begin{array}{llllll}
\min & \sum_{(i,j) \in E} c_{ij} x_{ij} & & & & \\
\text{s.t.} & \sum_{j \in V} x_{ij} & = & 1 & \forall i \in V & \text{(assign\_i)} \\
 & \sum_{i \in V} x_{ij} & = & 1 & \forall j \in V & \text{(assign\_j)} \\
 & u_i - u_j + 1 & \leq & (|V|-1)(1-x_{ij}) & \forall (i,j) \in V, i \neq 1, j \neq 1 & \text{(mtz)} \\
 & 2 \leq u_i & \leq & |V| & \forall i \in \{2, .., |V|\}, & \\
 & x_{ij} \in \{0,1\} & & & \forall (i,j) \in E &
\end{array}$$

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:

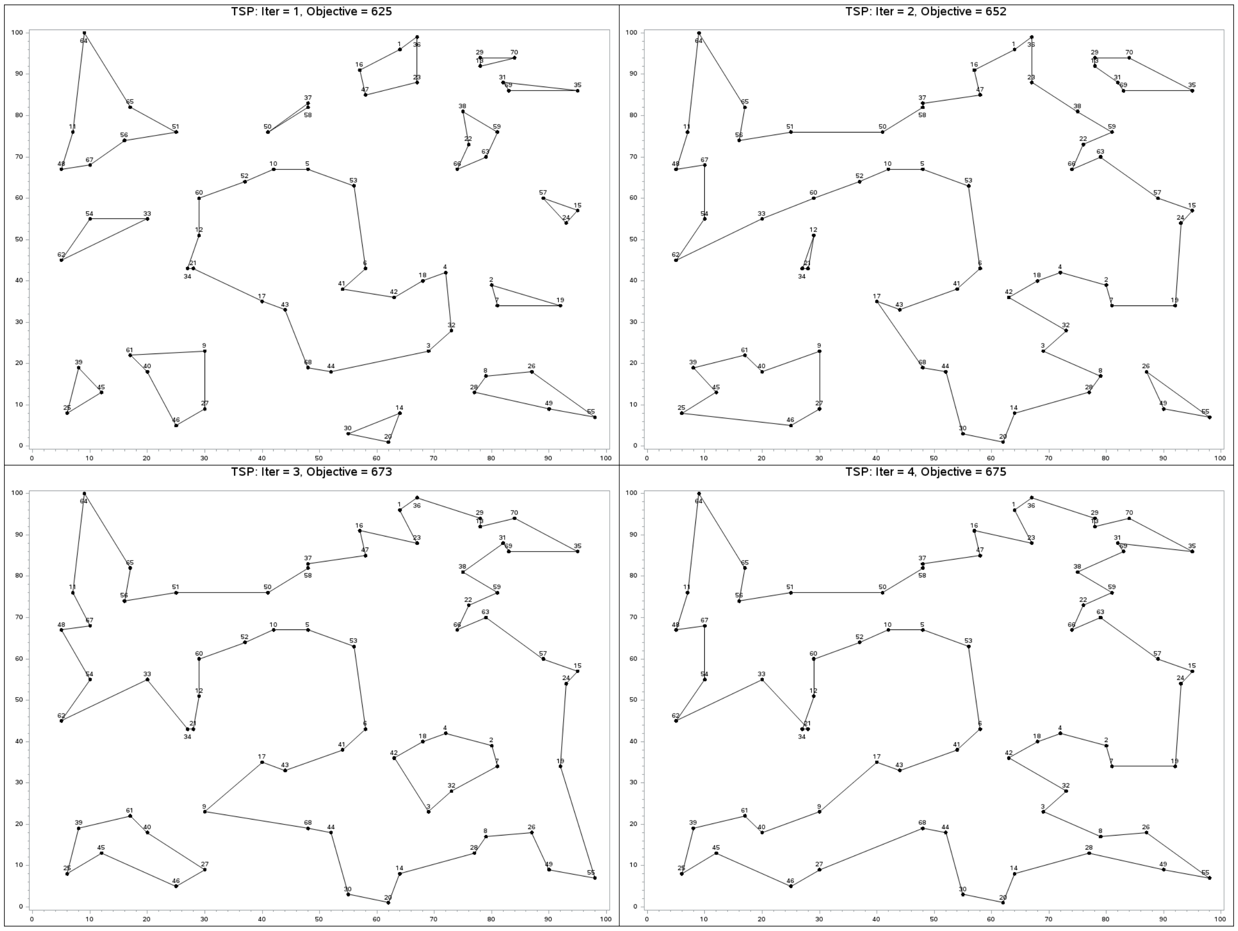
```
%macro plotTSP;
   %annomac;
   %do i = 1 %to &numiters;
      /* create annotate data set to draw subtours */
      data anno(drop=iter xi yi xj yj);
         %SYSTEM(2, 2, 2);
         set solData(keep=iter xi yi xj yj);
         where iter = &i;
         %LINE(xi, yi, xj, yj, *, 1, 1);
      run;

      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, 1999.

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.

Subject Index

Syntax Index