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

Chapter 13
The OPTMILP Procedure

Contents

Overview: OPTMILP Procedure

The OPTMILP procedure solves general mixed integer linear programs (MILPs).

A standard mixed integer linear program has the formulation

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

where

$\mathbf{x}$	$\in$	$\mathbb{R}^n$	is the vector of structural variables
$\mathbf{A}$	$\in$	$\mathbb{R}^{m \times n}$	is the matrix of technological coefficients
$\mathbf{c}$	$\in$	$\mathbb{R}^n$	is the vector of objective function coefficients
$\mathbf{b}$	$\in$	$\mathbb{R}^m$	is the vector of constraints right-hand sides (RHS)
$\mathbf{l}$	$\in$	$\mathbb{R}^n$	is the vector of lower bounds on variables
$\mathbf{u}$	$\in$	$\mathbb{R}^n$	is the vector of upper bounds on variables
$\mathcal{S}$			is a nonempty subset of the set $\{1 \ldots, n\}$ of indices

The OPTMILP procedure implements a linear-programming-based branch-and-cut algorithm. This divide-and-conquer approach attempts to solve the original problem by solving linear programming relaxations of a sequence of smaller subproblems. The OPTMILP procedure also implements advanced techniques such as presolving, generating cutting planes, and applying primal heuristics to improve the efficiency of the overall algorithm.

The OPTMILP procedure requires a mixed integer linear program to be specified using a SAS data set that adheres to the mathematical programming system (MPS) format, a widely accepted format in the optimization community. Chapter 17 discusses the MPS format in detail. It is also possible to input an incumbent solution in MPS format; see the section "Warm Start" on page 643 for details.

You can use the MPSOUT= option to convert data sets that are formatted for the LP procedure into MPS-format SAS data sets. The option is available in the LP, INTPOINT, and NETFLOW procedures. For details about this option, see Chapter 4, "The LP Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), Chapter 3, "The INTPOINT Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*), and Chapter 5, "The NETFLOW Procedure" (*SAS/OR User's Guide: Mathematical Programming Legacy Procedures*).

The OPTMILP procedure provides various control options and solution strategies. In particular, you can enable, disable, or set levels for the advanced techniques previously mentioned.

The OPTMILP procedure outputs an optimal solution or the best feasible solution found, if any, in SAS data sets. This enables you to generate solution reports and perform additional analyses by using SAS software.

Getting Started: OPTMILP Procedure

The following example illustrates the use of the OPTMILP procedure to solve mixed integer linear programs. For more examples, see the section "Examples: OPTMILP Procedure" on page 657. Suppose you want to solve the following problem:

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

The corresponding MPS-format SAS data set follows:

```
data ex_mip;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
NAME            .     EX_MIP        .    .         .
ROWS            .     .             .    .         .
N          COST       .             .    .         .
G          R1         .             .    .         .
L          R2         .             .    .         .
L          R3         .             .    .         .
COLUMNS         .     .             .    .         .
.          MARK00 'MARKER'          .    'INTORG'  .
.          X1     COST              2    R2        1
.          X1     R3                1    .         .
.          X2     COST             -3    R1        -2
.          X2     R2                1    R3        2
.          X3     COST             -4    R1        -3
.          X3     R2                2    R3        3
.          MARK01 'MARKER'          .    'INTEND'  .
RHS             .     .             .    .         .
.          RHS    R1               -5    R2        4
.          RHS    R3                7    .         .
ENDATA          .     .             .    .         .
;
```

You can also create this SAS data set from an MPS-format flat file (ex_mip.mps) by using the following SAS macro:

```
%mps2sasd(mpsfile = "ex_mip.mps", outdata = ex_mip);
```

This problem can be solved by using the following statement to call the OPTMILP procedure:

```
proc optmilp data = ex_mip
   objsense   = min
   primalout  = primal_out
   dualout    = dual_out
   presolver  = automatic
   heuristics = automatic;
run;
```

The DATA= option names the MPS-format SAS data set that contains the problem data. The OBJSENSE= option specifies whether to maximize or minimize the objective function. The PRIMALOUT= option names the SAS data set to contain the optimal solution or the best feasible solution found by the solver. The

DUALOUT= option names the SAS data set to contain the constraint activities. The PRESOLVER= and HEURISTICS= options specify the levels for presolving and applying heuristics, respectively. In this example, each option is set to its default value AUTOMATIC, meaning that the solver automatically determines the appropriate levels for presolve and heuristics.

The optimal integer solution and its corresponding constraint activities, stored in the data sets primal_out and dual_out, respectively, are displayed in Figure 13.1 and Figure 13.2.

Figure 13.1 Optimal Solution

The OPTMILP Procedure
Primal Integer Solution

Obs	Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value
1	COST	RHS	X1	B	2	0	1	0
2	COST	RHS	X2	B	-3	0	1	1
3	COST	RHS	X3	B	-4	0	1	1

Figure 13.2 Constraint Activities

The OPTMILP Procedure
Constraint Information

Obs	Objective Function ID	RHS ID	Constraint Name	Constraint Type	Constraint RHS	Constraint Lower Bound	Constraint Upper Bound	Constraint Activity
1	COST	RHS	R1	G	-5	.	.	-5
2	COST	RHS	R2	L	4	.	.	3
3	COST	RHS	R3	L	7	.	.	5

The solution summary stored in the macro variable _OROPTMILP_ can be viewed by issuing the following statement:

```
%put &_OROPTMILP_;
```

This produces the output shown in Figure 13.3.

Figure 13.3 Macro Output

```
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=-7 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0 BEST_BOUND=-7 NODES=0 ITERATIONS=0 PRESOLVE_TIME=0.35
SOLUTION_TIME=0.35
```

See the section "Data Input and Output" on page 641 for details about the type and status codes displayed for variables and constraints.

Syntax: OPTMILP Procedure

The following statements are available in the OPTMILP procedure:

PROC OPTMILP *< options >* **;**
    DECOMP *< options >* **;**
    DECOMP_MASTER *< options >* **;**
    DECOMP_MASTER_IP *< options >* **;**
    DECOMP_SUBPROB *< options >* **;**
    PERFORMANCE *< performance-options >* **;**
    TUNER *< tuner-options >* **;**

Functional Summary

Table 13.1 summarizes the options available for the OPTMILP procedure, classified by function.

Table 13.1   Options for the OPTMILP Procedure

Description	Option
Data Set Options	
Specifies the input data set	DATA=
Specifies the constraint activities output data set	DUALOUT=
Specifies whether the MILP model is a maximization or minimization problem	OBJSENSE=
Specifies the primal solution input data set (warm start)	PRIMALIN=
Specifies the primal solution output data set	PRIMALOUT=
Presolve Option	
Specifies the type of presolve	PRESOLVER=
Control Options	
Specifies the stopping criterion based on absolute objective gap	ABSOBJGAP=
Specifies the cutoff value for node removal	CUTOFF=
Emphasizes feasibility or optimality	EMPHASIS=
Specifies the maximum violation on variables and constraints	FEASTOL=
Specifies the maximum allowed difference between an integer variable's value and an integer	INTTOL=
Specifies the frequency of printing the node log	LOGFREQ=
Specifies the detail of solution progress printed in log	LOGLEVEL=
Specifies the maximum number of nodes to be processed	MAXNODES=
Specifies the maximum number of solutions to be found	MAXSOLS=
Specifies the time limit for the optimization process	MAXTIME=
Specifies the tolerance used in determining the optimality of nodes in the branch-and-bound tree	OPTTOL=
Toggles ODS output	PRINTLEVEL=
Specifies the probing level	PROBE=
Specifies the stopping criterion based on relative objective gap	RELOBJGAP=
Specifies the scale of the problem matrix	SCALE=
Specifies the initial seed for the random number generator	SEED=

Table 13.1 (continued)

Description	Option
Specifies the stopping criterion based on target objective value	TARGET=
Specifies whether time units are CPU time or real time	TIMETYPE=
Heuristics Option	
Specifies the primal heuristics level	HEURISTICS=
Search Options	
Specifies the level of conflict search	CONFLICTSEARCH=
Specifies the node selection strategy	NODESEL=
Enables use of variable priorities	PRIORITY=
Specifies the restarting strategy	RESTARTS=
Specifies the number of simplex iterations performed on each variable in strong branching strategy	STRONGITER=
Specifies the number of candidates for strong branching	STRONGLEN=
Specifies the level of symmetry detection	SYMMETRY=
Specifies the rule for selecting branching variable	VARSEL=
Cut Options	
Specifies the cut level for all cuts	ALLCUTS=
Specifies the clique cut level	CUTCLIQUE=
Specifies the flow cover cut level	CUTFLOWCOVER=
Specifies the flow path cut level	CUTFLOWPATH=
Specifies the Gomory cut level	CUTGOMORY=
Specifies the generalized upper bound (GUB) cover cut level	CUTGUB=
Specifies the implied bounds cut level	CUTIMPLIED=
Specifies the knapsack cover cut level	CUTKNAPSACK=
Specifies the lift-and-project cut level	CUTLAP=
Specifies the mixed lifted 0-1 cut level	CUTMILIFTED=
Specifies the mixed integer rounding (MIR) cut level	CUTMIR=
Specifies the multicommodity network flow cut level	CUTMULTICOMMODITY=
Specifies the row multiplier factor for cuts	CUTSFACTOR=
Specifies the overall cut aggressiveness	CUTSTRATEGY=
Specifies the zero-half cut level	CUTZEROHALF=

PROC OPTMILP Statement

> **PROC OPTMILP** < *options* > ;

You can specify the following options in the PROC OPTMILP statement.

Data Set Options

DATA=*SAS-data-set*
> specifies the input data set that corresponds to the MILP model. If this option is not specified, PROC OPTMILP uses the most recently created SAS data set. See Chapter 17, "The MPS-Format SAS Data Set," for more details about the input data set.

DUALOUT=*SAS-data-set*

DOUT=*SAS-data-set*

    specifies the output data set to contain the constraint activities.

OBJSENSE=MIN | **MAX**

    specifies whether the MILP model is a minimization or a maximization problem. You can use OBJSENSE=MIN for a minimization problem and OBJSENSE=MAX for a maximization problem. Alternatively, you can specify the objective sense in the input data set. This option supersedes the objective sense specified in the input data set. If the objective sense is not specified anywhere, then PROC OPTMILP interprets and solves the MILP as a minimization problem.

PRIMALIN=*SAS-data-set*

    enables you to input a warm start solution in a SAS data set. PROC OPTMILP validates both the data set and the solution stored in the data set. If the data set is not valid, then the PRIMALIN= data are ignored. If the solution stored in a valid PRIMALIN= data set is a feasible integer solution, then it provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution stored in a valid PRIMALIN= data set is infeasible, contains missing values, or contains fractional values for integer variables, PROC OPTMILP tries to repair the solution with a number of specialized repair heuristics. See the section "Warm Start" on page 643 for details.

PRIMALOUT=*SAS-data-set*

POUT=*SAS-data-set*

    specifies the output data set for the primal solution. This data set contains the primal solution information. See the section "Data Input and Output" on page 641 for details.

Presolve Option

PRESOLVER=*number* | *string*

    specifies a presolve *string* or its corresponding value *number*, as listed in Table 13.2.

Table 13.2 Values for PRESOLVER= Option

number	string	Description
−1	AUTOMATIC	Applies the default level of presolve processing
0	NONE	Disables presolver
1	BASIC	Performs minimal presolve processing
2	MODERATE	Applies a higher level of presolve processing
3	AGGRESSIVE	Applies the highest level of presolve processing

The default value is AUTOMATIC.

Control Options

ABSOBJGAP=*number*

    specifies a stopping criterion. When the absolute difference between the best integer objective and the best bound on the objective function value becomes smaller than the value of *number*, the procedure stops. The value of *number* can be any nonnegative number; the default value is 1E–6.

CUTOFF=_number_

cuts off any nodes in a minimization (maximization) problem that have an objective value at or above (below) *number*. The value of *number* can be any number; the default value is the positive (negative) number that has the largest absolute value that can be represented in your operating environment.

EMPHASIS=_number_ | *string*

specifies a search emphasis *string* or its corresponding value *number* as listed in Table 13.3.

Table 13.3   Values for EMPHASIS= Option

number	*string*	**Description**
0	BALANCE	Performs a balanced search
1	OPTIMAL	Emphasizes optimality over feasibility
2	FEASIBLE	Emphasizes feasibility over optimality

The default value is BALANCE.

FEASTOL=_number_

specifies the tolerance that PROC OPTMILP uses to check the feasibility of a solution. This tolerance applies both to the maximum violation of bounds on variables and to the difference between the right-hand sides and left-hand sides of constraints. The value of *number* can be any value between 1E–4 and 1E–9, inclusive. However, the value of *number* cannot be larger than the integer feasibility tolerance. If the value of *number* is larger than the value of the INTTOL= option, then PROC OPTMILP sets FEASTOL= to the value of INTTOL=. The default value is 1E–6.

If PROC OPTMILP fails to find a feasible solution within this tolerance but does find a solution that has some violation, then the procedure stops with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMILP_" on page 654).

INTTOL=_number_

specifies the amount by which an integer variable value can differ from an integer and still be considered integer feasible. The value of *number* can be any number between 1E–9 and 0.5, inclusive. PROC OPTMILP attempts to find an optimal solution whose integer infeasibility is less than *number*. The default value is 1E–5.

If the best solution that PROC OPTMILP finds has an integer infeasibility larger than the value of *number*, then PROC OPTMILP stops with a solution status of OPTIMAL_COND (see the section "Macro Variable _OROPTMILP_" on page 654).

LOGFREQ=_number_

PRINTFREQ=_number_

specifies how often information is printed in the node log. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is 100. If *number* is set to 0, then the node log is disabled. If *number* is positive, then an entry is made in the node log at the first node, at the last node, and at intervals dictated by the value of *number*. An entry is also made each time a better integer solution is found.

LOGLEVEL=*number | string*

PRINTLEVEL2=*number | string*

> controls the amount of information displayed in the SAS log by the solver, from a short description of presolve information and summary to details at each node. Table 13.4 describes the valid values for this option.

<div align="center">

Table 13.4   Values for LOGLEVEL= Option

</div>

number	string	Description
0	NONE	Turns off all solver-related messages in the SAS log
1	BASIC	Displays a solver summary after stopping
2	MODERATE	Prints a solver summary and a node log by using the interval dictated by the LOGFREQ= option
3	AGGRESSIVE	Prints a detailed solver summary and a node log by using the interval dictated by the LOGFREQ= option

> The default value is MODERATE.

MAXNODES=*number*

> specifies the maximum number of branch-and-bound nodes to be processed. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is $2^{31} - 1$.

MAXSOLS=*number*

> specifies a stopping criterion. If *number* solutions have been found, then the procedure stops. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is $2^{31} - 1$.

MAXTIME=*t*

> specifies an upper limit of *t* seconds of time for reading in the data and performing the optimization process. The value of the TIMETYPE= option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OPTTOL=*number*

> specifies the tolerance used to determine the optimality of nodes in the branch-and-bound tree. The value of *number* can be any value between (and including) 1E–4 and 1E–9. The default value is 1E–6.

PRINTLEVEL=0 | 1 | 2

> specifies whether a summary of the problem and solution should be printed. If PRINTLEVEL=1, then the Output Delivery System (ODS) tables ProblemSummary, SolutionSummary, and PerformanceInfo are produced and printed. If PRINTLEVEL=2, then the same tables are produced and printed along with an additional table called ProblemStatistics. If PRINTLEVEL=0, then no ODS tables are produced or printed. The default value is 1.

> For details about the ODS tables created by PROC OPTMILP, see the section "ODS Tables" on page 650.

PROBE=*number | string*

specifies a probing *string* or its corresponding value *number*, as listed in Table 13.5.

Table 13.5 Values for PROBE= Option

number	string	Description
–1	AUTOMATIC	Uses the probing strategy determined by PROC OPTMILP
0	NONE	Disables probing
1	MODERATE	Uses the probing moderately
2	AGGRESSIVE	Uses the probing aggressively

The default value is AUTOMATIC. See the section "Presolve and Probing" on page 646 for more information.

RELOBJGAP=*number*

specifies a stopping criterion based on the best integer objective (BestInteger) and the best bound on the objective function value (BestBound). The relative objective gap is equal to

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

When this value becomes smaller than the specified gap size *number*, the procedure stops. The value of *number* can be any nonnegative number; the default value is 1E–4.

SCALE=*number | string*

indicates whether to scale the problem matrix. SCALE= can take either of the values AUTOMATIC (–1) and NONE (0). SCALE=AUTOMATIC scales the matrix as determined by PROC OPTMILP; SCALE=NONE disables scaling. The default value is AUTOMATIC.

SEED=*number*

specifies the initial seed of the random number generator. This option affects the perturbation in the simplex solvers; thus it might result in a different optimal solution and a different solver path. This option usually has a significant, but unpredictable, effect on the solution time. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of the seed is 100.

TARGET=*number*

specifies a stopping criterion for minimization (maximization) problems. If the best integer objective is better than or equal to *number*, the procedure stops. The value of *number* can be any number; the default value is the negative (positive) number that has the largest absolute value representable in your operating environment.

TIMETYPE=*number | string*

specifies whether CPU time or real time is used for the MAXTIME= option and the _OROPTMILP_ macro variable in a PROC OPTMILP call. Table 13.6 describes the valid values of the TIMETYPE= option.

Table 13.6 Values for TIMETYPE= Option

number	string	Description
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

The default value of the TIMETYPE= option depends on the algorithm used and on various options. When the solver is used with distributed or multithreaded processing, then by default TIMETYPE= REAL. Otherwise, by default TIMETYPE= CPU. Table 13.7 describes the detailed logic for determining the default; the first context in the table that applies determines the default value. The NTHREADS= and NODES= options are specified in the PERFORMANCE statement. For more information about the NTHREADS= and NODES= options, see the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Table 13.7 Default Value for TIMETYPE= Option

Context	Default
NODES= value is nonzero for the decomposition algorithm	REAL
NTHREADS= value is greater than 1	REAL
NTHREADS= 1	CPU

Heuristics Option

HEURISTICS=*number* | *string*

controls the level of primal heuristics applied by PROC OPTMILP. This level determines how frequently primal heuristics are applied during the branch-and-bound tree search. It also affects the maximum number of iterations allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. The values of *string* and the corresponding values of *number* are listed in Table 13.8.

Table 13.8 Values for HEURISTICS= Option

number	*string*	Description
–1	AUTOMATIC	Applies the default level of heuristics, similar to MODERATE
0	NONE	Disables all primal heuristics
1	BASIC	Applies basic primal heuristics at low frequency
2	MODERATE	Applies most primal heuristics at moderate frequency
3	AGGRESSIVE	Applies all primal heuristics at high frequency

Setting HEURISTICS=NONE does not disable the heuristics that repair an infeasible input solution that is specified in a PRIMALIN= data set.

The default value of the HEURISTICS= option is AUTOMATIC. For details about primal heuristics, see the section "Primal Heuristics" on page 648.

Search Options

CONFLICTSEARCH=*number* | *string*

specifies the level of conflict search performed by PROC OPTMILP. Conflict search is used to find clauses resulting from infeasible subproblems that arise in the search tree. The values of *string* and the corresponding values of *number* are listed in Table 13.9.

Table 13.9 Values for CONFLICTSEARCH= Option

number	string	Description
–1	AUTOMATIC	Performs conflict search based on a strategy determined by PROC OPTMILP
0	NONE	Disables conflict search
1	MODERATE	Performs a moderate conflict search
2	AGGRESSIVE	Performs an aggressive conflict search

The default value is AUTOMATIC.

NODESEL=*number* | *string*

specifies the node selection strategy *string* or its corresponding value *number*, as listed in Table 13.10.

Table 13.10 Values for NODESEL= Option

number	string	Description
–1	AUTOMATIC	Uses automatic node selection
0	BESTBOUND	Chooses the node with the best relaxed objective (best-bound-first strategy)
1	BESTESTIMATE	Chooses the node with the best estimate of the integer objective value (best-estimate-first strategy)
2	DEPTH	Chooses the most recently created node (depth-first strategy)

The default value is AUTOMATIC. For details about node selection, see the section "Node Selection" on page 645.

PRIORITY=0 | 1

indicates whether to use specified branching priorities for integer variables. PRIORITY=0 ignores variable priorities; PRIORITY=1 uses priorities when they exist. The default value is 1. See the section "Branching Priorities" on page 646 for details.

RESTARTS=*number* | *string*

specifies the strategy for restarting the processing of the root node. The values of *string* and the corresponding values of *number* are listed in Table 13.11.

Table 13.11 Values for RESTARTS= Option

number	string	Description
–1	AUTOMATIC	Uses a restarting strategy determined by PROC OPTMILP
0	NONE	Disables restarting
1	BASIC	Uses a basic restarting strategy
2	MODERATE	Uses a moderate restarting strategy
3	AGGRESSIVE	Uses an aggressive restarting strategy

The default value is AUTOMATIC.

STRONGITER=*number* | **AUTOMATIC**

specifies the number of simplex iterations performed for each variable in the candidate list when using the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, PROC OPTMILP uses the default value; this value is calculated automatically.

STRONGLEN=*number* | **AUTOMATIC**

specifies the number of candidates used when performing the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, PROC OPTMILP uses the default value; this value is calculated automatically.

SYMMETRY=*number* | *string*

specifies the level of symmetry detection. Symmetry detection identifies groups of equivalent decision variables and uses this information to solve the problem more efficiently. The values of *string* and the corresponding values of *number* are listed in Table 13.12.

Table 13.12   Values for SYMMETRY= Option

number	*string*	**Description**
–1	AUTOMATIC	Performs symmetry detection based on a strategy that is determined by PROC OPTMILP
0	NONE	Disables symmetry detection
1	BASIC	Performs a basic symmetry detection
2	MODERATE	Performs a moderate symmetry detection
3	AGGRESSIVE	Performs an aggressive symmetry detection

The default value is AUTOMATIC. For more information about symmetry detection, see (Ostrowski 2008).

VARSEL=*number* | *string*

specifies the rule for selecting the branching variable. The values of *string* and the corresponding values of *number* are listed in Table 13.13.

Table 13.13   Values for VARSEL= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses automatic branching variable selection
0	MAXINFEAS	Chooses the variable with maximum infeasibility
1	MININFEAS	Chooses the variable with minimum infeasibility
2	PSEUDO	Chooses a branching variable based on pseudocost
3	STRONG	Uses strong branching variable selection strategy

The default value is AUTOMATIC. For details about variable selection, see the section "Variable Selection" on page 645.

Cut Options

Table 13.14 describes the *string* and *number* values for the cut options in PROC OPTMILP.

Table 13.14 Values for Individual Cut Options

number	string	Description
–1	AUTOMATIC	Generates cutting planes based on a strategy determined by PROC OPTMILP
0	NONE	Disables generation of cutting planes
1	MODERATE	Uses a moderate cut strategy
2	AGGRESSIVE	Uses an aggressive cut strategy

You can specify the CUTSTRATEGY= option to set the overall aggressiveness of the cut generation in PROC OPTMILP. Alternatively, you can use the ALLCUTS= option to set all cut types to the same level. You can override the ALLCUTS= value by using the options that correspond to particular cut types. For example, if you want PROC OPTMILP to generate only Gomory cuts, specify ALLCUTS=NONE and CUTGOMORY=AUTOMATIC. If you want to generate all cuts aggressively but generate no lift-and-project cuts, set ALLCUTS=AGGRESSIVE and CUTLAP=NONE.

ALLCUTS=_number_ | *string*

> provides a shorthand way of setting all the cuts-related options in one setting. In other words, ALLCUTS=*number* is equivalent to setting each of the individual cuts parameters to the same value *number*. Thus, ALLCUTS=–1 has the effect of setting CUTCLIQUE=–1, CUTFLOWCOVER=–1, CUTFLOWPATH=–1, ..., CUTMULTICOMMODITY=–1, and CUTZEROHALF=–1. Table 13.14 lists the values that can be assigned to *string* and *number*. In addition, you can override levels for individual cuts with the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIMPLIED=, CUTKNAPSACK=, CUTLAP=, CUTMILIFTED=, CUTMIR=, CUTMULTICOMMODITY=, and CUTZEROHALF= options. If the ALLCUTS= option is not specified, all the cuts-related options are either set to their individually specified values (if the corresponding option is specified) or to their default values (if that option is not specified).

CUTCLIQUE=_number_ | *string*

> specifies the level of clique cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTCLIQUE= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTFLOWCOVER=_number_ | *string*

> specifies the level of flow cover cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTFLOWCOVER= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTFLOWPATH=_number_ | *string*

> specifies the level of flow path cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTFLOWPATH= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTGOMORY=*number | string*

specifies the level of Gomory cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTGOMORY= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTGUB=*number | string*

specifies the level of generalized upper bound (GUB) cover cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTGUB= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTIMPLIED=*number | string*

specifies the level of implied bound cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTIMPLIED= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTKNAPSACK=*number | string*

specifies the level of knapsack cover cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTKNAPSACK= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTLAP=*number | string*

specifies the level of lift-and-project (LAP) cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTLAP= option overrides the ALLCUTS= option. The default value is NONE.

CUTMILIFTED=*number | string*

specifies the level of mixed lifted 0-1 cuts that are generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *option* and *num*. The CUTMILIFTED= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTMIR=*number | string*

specifies the level of mixed integer rounding (MIR) cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTMIR= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTMULTICOMMODITY=*number | string*

specifies the level of multicommodity network flow cuts generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*. The CUTMULTICOMMODITY= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

CUTSFACTOR=*number*

specifies a row multiplier factor for cuts. The number of cuts that are added is limited to *number* times the original number of rows. The value of *number* can be any nonnegative number less than or equal to 100; the default value is automatically calculated by PROC OPTMILP.

CUTSTRATEGY=*number | string*

CUTS=*number | string*

specifies the overall aggressiveness of the cut generation in the solver. Setting a nondefault value adjusts a number of cut parameters such that the cut generation is basic, moderate, or aggressive compared to the default value.

CUTZEROHALF=*number | string*
> specifies the level of zero-half cuts that are generated by PROC OPTMILP. Table 13.14 lists the values that can be assigned to *string* and *number*.The CUTZEROHALF= option overrides the ALLCUTS= option. The default value is AUTOMATIC.

Decomposition Algorithm Statements

The following statements are available for the decomposition algorithm in the OPTMILP procedure:

> **DECOMP** < *options* > **;**

> **DECOMP_MASTER** < *options* > **;**

> **DECOMP_MASTER_IP** < *options* > **;**

> **DECOMP_SUBPROB** < *options* > **;**

For more information about these statements, see Chapter 15, "The Decomposition Algorithm."

PERFORMANCE Statement

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

The PERFORMANCE statement specifies *performance-options* for single-machine mode and distributed mode, and requests detailed performance results of the OPTMILP procedure.

You can also use the PERFORMANCE statement to control whether the OPTMILP procedure executes in single-machine or distributed mode. The decomposition algorithm and the option tuner can be run in both single-machine and distributed modes. The parallel branch-and-cut algorithm can be run only in single-machine mode.

The PERFORMANCE statement is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

NOTE: Distributed mode requires SAS High-Performance Optimization.

TUNER Statement

> **TUNER** < *performance-options* > **;**

The TUNER statement invokes the OPTMILP option tuner. The option tuner is a tool that enables you to explore alternative (and potentially better) option configurations for your optimization problems. For more information about this feature, see Chapter 16, "The OPTMILP Option Tuner."

Details: OPTMILP Procedure

Data Input and Output

This subsection describes the PRIMALIN= data set required to warm start PROC OPTMILP, in addition to the PRIMALOUT= and DUALOUT= data sets.

Definitions of Variables in the PRIMALIN= Data Set

The PRIMALIN= data set has two required variables defined as follows:

VAR
> specifies the variable (column) names of the problem. The values should match the column names in the DATA= data set for the current problem.

VALUE
> specifies the solution value for each variable in the problem.

NOTE: If PROC OPTMILP produces a feasible solution, the primal output data set from that run can be used as the PRIMALIN= data set for a subsequent run, provided that the variable names are the same. If this input solution is not feasible for the subsequent run, the solver automatically tries to repair it. See the section "Warm Start" on page 643 for more details.

Definitions of Variables in the PRIMALOUT= Data Set

PROC OPTMILP stores the current best integer feasible solution of the problem in the data set specified by the PRIMALOUT= option. The variables in this data set are defined as follows:

_OBJ_ID_
> specifies the identifier of the objective function.

_RHS_ID_
> specifies the identifier of the right-hand side.

VAR
> specifies the variable (column) names.

TYPE
> specifies the variable type. _TYPE_ can take one of the following values:

> C   continuous variable

> I   general integer variable

> B   binary variable (0 or 1)

OBJCOEF
>   specifies the coefficient of the variable in the objective function.

LBOUND
>   specifies the lower bound on the variable.

UBOUND
>   specifies the upper bound on the variable.

VALUE
>   specifies the value of the variable in the current solution.

Definitions of the DUALOUT= Data Set Variables

The DUALOUT= data set contains the constraint activities that correspond to the primal solution in the PRIMALOUT= data set. Information about additional objective rows of the MILP problem is not included. The variables in this data set are defined as follows:

_OBJ_ID_
>   specifies the identifier of the objective function from the input data set.

_RHS_ID_
>   specifies the identifier of the right-hand side from the input data set.

ROW
>   specifies the constraint (row) name.

TYPE
>   specifies the constraint type. _TYPE_ can take one of the following values:
>
>   L   "less than or equal" constraint
>
>   E   equality constraint
>
>   G   "greater than or equal" constraint
>
>   R   ranged constraint (both "less than or equal" and "greater than or equal")

RHS
>   specifies the value of the right-hand side of the constraint. It takes a missing value for a ranged constraint.

_L_RHS_
>   specifies the lower bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_U_RHS_
>   specifies the upper bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

ACTIVITY
>   specifies the activity of a constraint for a given primal solution. In other words, the value of _ACTIVITY_ for the ith constraint is equal to $a_i^T x$, where a_i refers to the ith row of the constraint matrix and x denotes the vector of the current primal solution.

Warm Start

PROC OPTMILP enables you to input a warm start solution by using the PRIMALIN= option. PROC OPTMILP checks that the decision variables named in _VAR_ are the same as those in the MPS-format SAS data set. If they are not the same, PROC OPTMILP issues a warning and ignores the input solution. PROC OPTMILP also checks whether the solution is infeasible, contains missing values, or contains fractional values for integer variables. If this is the case, PROC OPTMILP attempts to repair the solution with a number of specialized repair heuristics. The success of the attempt largely depends both on the specific model and on the proximity between the input solution and an integer feasible solution. An infeasible input solution can be considered a hint for PROC OPTMILP that might or might not help to solve the problem.

An integer feasible or repaired input solution provides an incumbent solution in addition to an upper (min) or lower (max) bound for the branch-and-bound algorithm. PROC OPTMILP uses the input solution to reduce the search space and to guide the search process. When it is difficult to find a good integer feasible solution for a problem, warm start can reduce solution time significantly.

Branch-and-Bound Algorithm

The branch-and-bound algorithm, first proposed by Land and Doig (1960), is an effective approach to solving mixed integer linear programs. The following discussion outlines the approach and explains how PROC OPTMILP enhances the basic algorithm by using several advanced techniques.

The branch-and-bound algorithm solves a mixed integer linear program by dividing the search space and generating a sequence of subproblems. The search space of a mixed integer linear program can be represented by a tree. Each node in the tree is identified with a subproblem derived from previous subproblems on the path that leads to the root of the tree. The subproblem (MILP^0) associated with the root is identical to the original problem, which is called (MILP), given in the section "Overview: OPTMILP Procedure" on page 625.

The linear programming relaxation (LP^0) of (MILP^0) can be written as

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

The branch-and-bound algorithm generates subproblems along the nodes of the tree by using the following scheme. Consider $\bar{x}^0$, the optimal solution to (LP^0), which is usually obtained by using the dual simplex algorithm. If $\bar{x}_i^0$ is an integer for all $i \in \mathcal{S}$, then $\bar{x}^0$ is an optimal solution to (MILP). Suppose that for some $i \in \mathcal{S}$, $\bar{x}_i^0$ is nonintegral. In that case the algorithm defines two new subproblems (MILP^1) and (MILP^2), descendants of the parent subproblem (MILP^0). The subproblem (MILP^1) is identical to (MILP^0) except for the additional constraint

$$ x_i \leq \lfloor \bar{x}_i^0 \rfloor $$

and the subproblem (MILP^2) is identical to (MILP^0) except for the additional constraint

$$ x_i \geq \lceil \bar{x}_i^0 \rceil $$

The notation $\lfloor y \rfloor$ represents the largest integer that is less than or equal to y, and the notation $\lceil y \rceil$ represents the smallest integer that is greater than or equal to y. The two preceding constraints can be handled by modifying the bounds of the variable x_i rather than by explicitly adding the constraints to the constraint matrix. The two new subproblems do not have $\bar{x}^0$ as a feasible solution, but the integer solution to (MILP) must satisfy one of the preceding constraints. The two subproblems thus defined are called *active nodes* in the branch-and-bound tree, and the variable x_i is called the *branching variable*.

In the next step the branch-and-bound algorithm chooses one of the active nodes and attempts to solve the linear programming relaxation of that subproblem. The relaxation might be infeasible, in which case the subproblem is dropped (fathomed). If the subproblem can be solved and the solution is *integer feasible* (that is, x_i is an integer for all $i \in \mathcal{S}$), then its objective value provides an *upper bound* for the objective value in the minimization problem (MILP); if the solution is not integer feasible, then it defines two new subproblems. Branching continues in this manner until there are no active nodes. At this point the best integer solution found is an optimal solution for (MILP). If no integer solution has been found, then (MILP) is integer infeasible. You can specify other criteria to stop the branch-and-bound algorithm before it processes all the active nodes; see the section "Controlling the Branch-and-Bound Algorithm" on page 644 for details.

Upper bounds from integer feasible solutions can be used to *fathom* or *cut off* active nodes. Since the objective value of an optimal solution cannot be greater than an upper bound, active nodes with lower bounds higher than an existing upper bound can be safely deleted. In particular, if z is the objective value of the current best integer solution, then any active subproblems whose relaxed objective value is greater than or equal to z can be discarded.

It is important to realize that mixed integer linear programs are non-deterministic polynomial-time hard (NP-hard). Roughly speaking, this means that the effort required to solve a mixed integer linear program grows exponentially with the size of the problem. For example, a problem with 10 binary variables can generate in the worst case $2^{10} = 1,024$ nodes in the branch-and-bound tree. A problem with 20 binary variables can generate in the worst case $2^{20} = 1,048,576$ nodes in the branch-and-bound tree. Although it is unlikely that the branch-and-bound algorithm has to generate every single possible node, the need to explore even a small fraction of the potential number of nodes for a large problem can be resource-intensive.

A number of techniques can speed up the search progress of the branch-and-bound algorithm. Heuristics are used to find feasible solutions, which can improve the upper bounds on solutions of mixed integer linear programs. Cutting planes can reduce the search space and thus improve the lower bounds on solutions of mixed integer linear programs. When using cutting planes, the branch-and-bound algorithm is also called the *branch-and-cut algorithm*. Preprocessing can reduce problem size and improve problem solvability. PROC OPTMILP employs various heuristics, cutting planes, preprocessing, and other techniques, which you can control through corresponding options.

Controlling the Branch-and-Bound Algorithm

There are numerous strategies that can be used to control the branch-and-bound search (see Linderoth and Savelsbergh 1998, Achterberg, Koch, and Martin 2005). PROC OPTMILP implements the most widely used strategies and provides several options that enable you to direct the choice of the next active node and of the branching variable. In the discussion that follows, let (LP^k) be the linear programming relaxation of subproblem (MILP^k). Also, let

$$f_i(k) = \bar{x}_i^k - \lfloor \bar{x}_i^k \rfloor$$

where $\bar{x}^k$ is the optimal solution to the relaxation problem (LP^k) solved at node k.

Node Selection

The NODESEL= option specifies the strategy used to select the next active node. The valid keywords for this option are AUTOMATIC, BESTBOUND, BESTESTIMATE, and DEPTH. The following list describes the strategy associated with each keyword:

AUTOMATIC allows PROC OPTMILP to choose the best node selection strategy based on problem characteristics and search progress. This is the default setting.

BESTBOUND chooses the node with the smallest (or largest, in the case of a maximization problem) relaxed objective value. The best-bound strategy tends to reduce the number of nodes to be processed and can improve lower bounds quickly. However, if there is no good upper bound, the number of active nodes can be large. This can result in the solver running out of memory.

BESTESTIMATE chooses the node with the smallest (or largest, in the case of a maximization problem) objective value of the estimated integer solution. Besides improving lower bounds, the best-estimate strategy also attempts to process nodes that can yield good feasible solutions.

DEPTH chooses the node that is deepest in the search tree. Depth-first search is effective in locating feasible solutions, since such solutions are usually deep in the search tree. Compared to the costs of the best-bound and best-estimate strategies, the cost of solving LP relaxations is less in the depth-first strategy. The number of active nodes is generally small, but it is possible that the depth-first search will remain in a portion of the search tree with no good integer solutions. This occurrence is computationally expensive.

Variable Selection

The VARSEL= option specifies the strategy used to select the next branching variable. The valid keywords for this option are AUTOMATIC, MAXINFEAS, MININFEAS, PSEUDO, and STRONG. The following list describes the action taken in each case when $\bar{x}^k$, a relaxed optimal solution of (MILP^k), is used to define two active subproblems. In the following list, "INTTOL" refers to the value assigned using the INTTOL= option. For details about the INTTOL= option, see the section "Control Options" on page 631.

AUTOMATIC enables PROC OPTMILP to choose the best variable selection strategy based on problem characteristics and search progress. This is the default setting.

MAXINFEAS chooses as the branching variable the variable x_i such that i maximizes

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

$$\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

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

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

$$\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

PSEUDO   chooses as the branching variable the variable x_i such that i maximizes the weighted up and down pseudocosts. Pseudocost branching attempts to branch on significant variables first, quickly improving lower bounds. Pseudocost branching estimates significance based on historical information; however, this approach might not be accurate for future search.

STRONG   chooses as the branching variable the variable x_i such that i maximizes the estimated improvement in the objective value. Strong branching first generates a list of candidates, then branches on each candidate and records the improvement in the objective value. The candidate with the largest improvement is chosen as the branching variable. Strong branching can be effective for combinatorial problems, but it is usually computationally expensive.

Branching Priorities

In some cases, it is possible to speed up the branch-and-bound algorithm by branching on variables in a specific order. You can accomplish this in PROC OPTMILP by attaching branching priorities to the integer variables in your model.

You can set branching priorities for use by PROC OPTMILP in two ways. You can specify the branching priorities directly in the input MPS-format data set; see the section "BRANCH Section (Optional)" on page 847 for details. If you are constructing a model in PROC OPTMODEL, you can set branching priorities for integer variables by using the .priority suffix. More information about this suffix is available in the section "Integer Variable Suffixes" on page 137 in Chapter 5. For an example in which branching priorities are used, see Example 8.3.

Presolve and Probing

PROC OPTMILP includes a variety of presolve techniques to reduce problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix. Presolve might also modify the constraint coefficients to tighten the formulation of the problem.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels by specifying the PRESOLVER= option.

Probing is a technique that tentatively sets each binary variable to 0 or 1, then explores the logical consequences (Savelsbergh 1994). Probing can expedite the solution of a difficult problem by fixing variables and improving the model. However, probing is often computationally expensive and can significantly increase the solution time in some cases. You can enable probing at different levels by specifying the PROBE= option.

Cutting Planes

The feasible region of every linear program forms a *polyhedron*. Every polyhedron in n-space can be written as a finite number of half-spaces (equivalently, inequalities). In the notation used in this chapter, this polyhedron is defined by the set $\mathcal{Q} = \{x \in \mathbb{R}^n \mid Ax \leq b, l \leq x \leq u\}$. After you add the restriction that

some variables must be integral, the set of feasible solutions, $\mathcal{F} = \{x \in \mathcal{Q} \mid x_i \in \mathbb{Z} \ \forall i \in \mathcal{S}\}$, no longer forms a polyhedron.

The *convex hull* of a set X is the minimal convex set that contains X. In solving a mixed integer linear program, in order to take advantage of LP-based algorithms you want to find the convex hull, conv($\mathcal{F}$), of $\mathcal{F}$. If you can find conv($\mathcal{F}$) and describe it compactly, then you can solve a mixed integer linear program with a linear programming solver. This is generally very difficult, so you must be satisfied with finding an approximation. Typically, the better the approximation, the more efficiently the LP-based branch-and-bound algorithm can perform.

As described in the section "Branch-and-Bound Algorithm" on page 643, the branch-and-bound algorithm begins by solving the linear programming relaxation over the polyhedron $\mathcal{Q}$. Clearly, $\mathcal{Q}$ contains the convex hull of the feasible region of the original integer program; that is, conv($\mathcal{F}$) $\subseteq \mathcal{Q}$.

Cutting plane techniques are used to tighten the linear relaxation to better approximate 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 conv($\mathcal{F}$); that is, every integer feasible solution is feasible for the cut ($\pi x \leq \pi^0, \forall x \in \mathcal{F}$).

- The half-space does not contain the current solution $\bar{x}$; that is, $\bar{x}$ is not feasible for the cut ($\pi \bar{x} > \pi^0$).

Cutting planes were first made popular by Dantzig, Fulkerson, and Johnson (1954) in their work on the traveling salesman problem. The two major classifications of cutting planes are *generic cuts* and *structured cuts*. Generic cuts are based solely on algebraic arguments and can be applied to any relaxation of any integer program. Structured cuts are specific to certain structures that can be found in some relaxations of the mixed integer linear program. These structures are automatically discovered during the cut initialization phase of PROC OPTMILP. Table 13.15 lists the various types of cutting planes that are built into PROC OPTMILP. Included in each type are algorithms for numerous variations based on different relaxations and lifting techniques. For a survey of cutting plane techniques for mixed integer programming, see Marchand et al. (1999). For a survey of lifting techniques, see Atamturk (2004).

Table 13.15   Cutting Planes in PROC OPTMILP

Generic Cutting Planes	Structured Cutting Planes
Gomory mixed integer	Cliques
Lift-and-project	Flow cover
Mixed integer rounding	Flow path
Mixed lifted 0-1	Generalized upper bound cover
Zero-half	Implied bound
	Knapsack cover
	Multicommodity network flow

You can set levels for individual cuts by using the CUTCLIQUE=, CUTFLOWCOVER=, CUTFLOWPATH=, CUTGOMORY=, CUTGUB=, CUTIMPLIED=, CUTKNAPSACK=, CUTLAP=, CUTMIR=, CUTMULTI-COMMODITY=, and CUTZEROHALF= options. The valid levels for these options are given in Table 13.14.

The cut level determines the internal strategy used by PROC OPTMILP for generating the cutting planes. The strategy consists of several factors, including how frequently the cut search is called, the number of cuts allowed, and the aggressiveness of the search algorithms.

Sophisticated cutting planes, such as those included in PROC OPTMILP, can take a great deal of CPU time. Usually, the additional tightening of the relaxation helps speed up the overall process because it provides better bounds for the branch-and-bound tree and helps guide the LP solver toward integer solutions. In rare cases, shutting off cutting planes completely might lead to faster overall run times.

The default settings of PROC OPTMILP have been tuned to work well for most instances. However, problem-specific expertise might suggest adjusting one or more of the strategies. These options give you that flexibility.

Primal Heuristics

Primal heuristics, an important component of PROC OPTMILP, are applied during the branch-and-bound algorithm. They are used to find integer feasible solutions early in the search tree, thereby improving the upper bound for a minimization problem. Primal heuristics play a role that is complementary to cutting planes in reducing the gap between the upper and lower bounds, thus reducing the size of the branch-and-bound tree.

Applying primal heuristics in the branch-and-bound algorithm assists in the following areas:

- finding a good upper bound early in the tree search (this can lead to earlier fathoming, resulting in fewer subproblems to be processed)

- locating a reasonably good feasible solution when that is sufficient (sometimes a good feasible solution is the best the solver can produce within certain time or resource limits)

- providing upper bounds for some bound-tightening techniques

The OPTMILP procedure implements several heuristic methodologies. Some algorithms, such as rounding and iterative rounding (diving) heuristics, attempt to construct an integer feasible solution by using fractional solutions to the continuous relaxation at each node of the branch-and-cut tree. Other algorithms start with an incumbent solution and attempt to find a better solution within a neighborhood of the current best solution.

The HEURISTICS= option enables you to control the level of primal heuristics that are applied by PROC OPTMILP. This level determines how frequently primal heuristics are applied during the tree search. Some expensive heuristics might be disabled by the solver at less aggressive levels. Setting the HEURISTICS= option to a lower level also reduces the maximum number of iterations that are allowed in iterative heuristics. The valid values for this option are listed in Table 13.8.

Parallel Processing

You can run the branch-and-cut algorithm only in single-machine mode. In single-machine mode, the computation is executed by multiple threads on a single computer.

You can run the decomposition algorithm and option tuner in either single-machine or distributed mode. In distributed mode, the computation is executed on multiple computing nodes in a distributed computing environment.

NOTE: Distributed mode requires SAS High-Performance Optimization.

You can specify options that control parallel processing in the PERFORMANCE statement, which is documented in the section "PERFORMANCE Statement" on page 21 in Chapter 4, "Shared Concepts and Topics."

Node Log

The following information about the status of the branch-and-bound algorithm is printed in the node log:

Node	indicates the sequence number of the current node in the search tree.
Active	indicates the current number of active nodes in the branch-and-bound tree.
Sols	indicates the number of feasible solutions found so far.
BestInteger	indicates the best upper bound (assuming minimization) found so far.
BestBound	indicates the best lower bound (assuming minimization) found so far.
Gap	indicates the relative gap between BestInteger and BestBound, displayed as a percentage. If the relative gap is larger than 1,000, then the absolute gap is displayed. If no active nodes remain, the value of Gap is 0.
Time	indicates the elapsed real time.

The LOGFREQ= and LOGLEVEL= options can be used to control the amount of information printed in the node log. By default a new entry is included in the log at the first node, at the last node, and at 100-node intervals. A new entry is also included each time a better integer solution is found. The LOGFREQ= option enables you to change the interval between entries in the node log. Figure 13.4 shows a sample node log.

Figure 13.4 Sample Node Log

```
NOTE: The problem ex1data has 10 variables (0 binary, 10 integer, 0 free, 0
      fixed).
NOTE: The problem has 2 constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 20 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 2 variables and 0 constraints.
NOTE: The MILP presolver removed 4 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 8 variables, 2 constraints, and 16 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node   Active     Sols     BestInteger       BestBound        Gap     Time
             0        1        3      85.0000000     158.0000000     46.20%        0
             0        1        3      85.0000000      88.0955497      3.51%        0
             0        1        3      85.0000000      88.0955497      3.51%        0
             0        1        3      85.0000000      88.0955497      3.51%        0
NOTE: The MILP presolver is applied again.
             0        1        3      85.0000000      88.0955497      3.51%        0
             0        1        3      85.0000000      88.0626822      3.48%        0
             0        1        3      85.0000000      87.8820655      3.28%        0
             0        1        3      85.0000000      87.8539763      3.25%        0
             0        1        3      85.0000000      87.7208690      3.10%        0
             0        1        3      85.0000000      87.7180302      3.10%        0
             0        1        3      85.0000000      87.7133502      3.09%        0
             0        1        3      85.0000000      87.7128245      3.09%        0
NOTE: The MILP solver added 2 cuts with 8 cut coefficients at the root.
             3        1        4      86.0000000      87.3854749      1.59%        0
             4        0        5      87.0000000      87.0000000      0.00%        0
NOTE: Optimal.
NOTE: Objective = 87.
NOTE: The data set WORK.EX1SOLN has 10 observations and 8 variables.
```

ODS Tables

PROC OPTMILP creates three Output Delivery System (ODS) tables by default. The first table, Problem-Summary, is a summary of the input MILP problem. The second table, SolutionSummary, is a brief summary of the solution status. The third table, PerformanceInfo, is a summary of performance options. You can use ODS table names to select tables and create output data sets. For more information about ODS, see *SAS Output Delivery System: User's Guide*.

If you specify a value of 2 for the PRINTLEVEL= option, then the ProblemStatistics table is produced. This table contains information about the problem data. See the section "Problem Statistics" on page 654 for more information.

If you specify the DETAILS option in the PERFORMANCE statement, then the Timing table is produced.

Table 13.16 lists all the ODS tables that can be produced by the OPTMILP procedure, along with the statement and option specifications required to produce each table.

Table 13.16 ODS Tables Produced by PROC OPTMILP

ODS Table Name	Description	Statement	Option
ProblemSummary	Summary of the input MILP problem	PROC OPTMILP	PRINTLEVEL=1 (default)
SolutionSummary	Summary of the solution status	PROC OPTMILP	PRINTLEVEL=1 (default)
ProblemStatistics	Description of input problem data	PROC OPTMILP	PRINTLEVEL=2
PerformanceInfo	List of performance options and their values	PROC OPTMILP	PRINTLEVEL=1 (default)
Timing	Detailed solution timing	PERFORMANCE	DETAILS

A typical ProblemSummary table is shown in Figure 13.5.

Figure 13.5 Example PROC OPTMILP Output: Problem Summary

The OPTMILP Procedure

Problem Summary	
Problem Name	EX_MIP
Objective Sense	Minimization
Objective Function	COST
RHS	RHS
Number of Variables	3
Bounded Above	0
Bounded Below	0
Bounded Above and Below	3
Free	0
Fixed	0
Binary	3
Integer	0
Number of Constraints	3
LE (<=)	2
EQ (=)	0
GE (>=)	1
Range	0
Constraint Coefficients	8

A typical SolutionSummary table is shown in Figure 13.6.

Figure 13.6 Example PROC OPTMILP Output: Solution Summary

The OPTMILP Procedure

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	COST
Solution Status	Optimal
Objective Value	-7
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	0
Best Bound	-7
Nodes	0
Iterations	0
Presolve Time	0.00
Solution Time	0.00

You can create output data sets from these tables by using the ODS OUTPUT statement. The output data sets from the preceding example are displayed in Figure 13.7 and Figure 13.8, where you can also find variable names for the tables used in the ODS template of the OPTMILP procedure.

Figure 13.7 ODS Output Data Set: Problem Summary

Problem Summary

Obs	Label1	cValue1	nValue1
1	Problem Name	EX_MIP	.
2	Objective Sense	Minimization	.
3	Objective Function	COST	.
4	RHS	RHS	.
5			.
6	Number of Variables	3	3.000000
7	Bounded Above	0	0
8	Bounded Below	0	0
9	Bounded Above and Below	3	3.000000
10	Free	0	0
11	Fixed	0	0
12	Binary	3	3.000000
13	Integer	0	0
14			.
15	Number of Constraints	3	3.000000
16	LE (<=)	2	2.000000
17	EQ (=)	0	0
18	GE (>=)	1	1.000000
19	Range	0	0
20			.
21	Constraint Coefficients	8	8.000000

Figure 13.8 ODS Output Data Set: Solution Summary

Solution Summary

Obs	Label1	cValue1	nValue1
1	Solver	MILP	.
2	Algorithm	Branch and Cut	.
3	Objective Function	COST	.
4	Solution Status	Optimal	.
5	Objective Value	-7	-7.000000
6			.
7	Relative Gap	0	0
8	Absolute Gap	0	0
9	Primal Infeasibility	0	0
10	Bound Infeasibility	0	0
11	Integer Infeasibility	0	0
12			.
13	Best Bound	-7	-7.000000
14	Nodes	0	0
15	Iterations	0	0
16	Presolve Time	0.00	0.001000
17	Solution Time	0.00	0.001000

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of 10^9) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMILP procedure causes the ODS table ProblemStatistics to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 13.9 demonstrates the contents of the ODS table ProblemStatistics.

Figure 13.9 ODS Table ProblemStatistics

ProblemStatistics

Obs	Label1	cValue1	nValue1
1	Number of Constraint Matrix Nonzeros	8	8.000000
2	Maximum Constraint Matrix Coefficient	3	3.000000
3	Minimum Constraint Matrix Coefficient	1	1.000000
4	Average Constraint Matrix Coefficient	1.875	1.875000
5			.
6	Number of Objective Nonzeros	3	3.000000
7	Maximum Objective Coefficient	4	4.000000
8	Minimum Objective Coefficient	2	2.000000
9	Average Objective Coefficient	3	3.000000
10			.
11	Number of RHS Nonzeros	3	3.000000
12	Maximum RHS	7	7.000000
13	Minimum RHS	4	4.000000
14	Average RHS	5.3333333333	5.333333
15			.
16	Maximum Number of Nonzeros per Column	3	3.000000
17	Minimum Number of Nonzeros per Column	2	2.000000
18	Average Number of Nonzeros per Column	2.67	2.666667
19			.
20	Maximum Number of Nonzeros per Row	3	3.000000
21	Minimum Number of Nonzeros per Row	2	2.000000
22	Average Number of Nonzeros per Row	2.67	2.666667

The variable names in the ODS table ProblemStatistics are Label1, cValue1, and nValue1.

Macro Variable _OROPTMILP_

The OPTMILP procedure defines a macro variable named _OROPTMILP_. This variable contains a character string that indicates the status of the OPTMILP procedure upon termination. The various terms of the variable are interpreted as follows.

STATUS

indicates the solver status at termination. It can take one of the following values:

OK	The procedure terminated normally.
SYNTAX_ERROR	Incorrect syntax was used.
DATA_ERROR	The input data was inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem occurred in reading or writing data.
ERROR	The status cannot be classified into any of the preceding categories.

ALGORITHM

indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take one of the following values:

BAC	The branch-and-cut algorithm produced the solution data.
DECOMP	The decomposition algorithm produced the solution data.

SOLUTION_STATUS

indicates the solution status at termination. It can take one of the following values:

OPTIMAL	The solution is optimal.
OPTIMAL_AGAP	The solution is optimal within the absolute gap specified by the ABSOBJGAP= option.
OPTIMAL_RGAP	The solution is optimal within the relative gap specified by the RELOBJGAP= option.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances due to scaling or choice of a small INTTOL= value.
TARGET	The solution is not worse than the target specified by the TARGET= option.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
SOLUTION_LIM	The solver reached the maximum number of solutions specified by the MAXSOLS= option.
NODE_LIM_SOL	The solver reached the maximum number of nodes specified by the MAXNODES= option and found a solution.
NODE_LIM_NOSOL	The solver reached the maximum number of nodes specified by the MAXNODES= option and did not find a solution.
TIME_LIM_SOL	The solver reached the execution time limit specified by the MAXTIME= option and found a solution.

TIME_LIM_NOSOL	The solver reached the execution time limit specified by the MAXTIME= option and did not find a solution.
ABORT_SOL	The solver was stopped by the user but still found a solution.
ABORT_NOSOL	The solver was stopped by the user and did not find a solution.
OUTMEM_SOL	The solver ran out of memory but still found a solution.
OUTMEM_NOSOL	The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
FAIL_SOL	The solver stopped due to errors but still found a solution.
FAIL_NOSOL	The solver stopped due to errors and did not find a solution.

OBJECTIVE

indicates the objective value obtained by the solver at termination.

RELATIVE_GAP

indicates the relative gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the OPTMILP procedure. The relative gap is equal to

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

ABSOLUTE_GAP

indicates the absolute gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the OPTMILP procedure. The absolute gap is equal to $| \text{ BestInteger} - \text{BestBound } |$.

PRIMAL_INFEASIBILITY

indicates the maximum (absolute) violation of the primal constraints by the solution.

BOUND_INFEASIBILITY

indicates the maximum (absolute) violation by the solution of the lower or upper bounds (or both).

INTEGER_INFEASIBILITY

indicates the maximum (absolute) violation of the integrality of integer variables returned by the OPTMILP procedure.

BEST_BOUND

indicates the best bound on the objective function value at termination. A missing value indicates that the OPTMILP procedure was not able to obtain such a bound.

NODES

indicates the number of nodes enumerated by the OPTMILP procedure when using the branch-and-bound algorithm.

ITERATIONS

indicates the number of simplex iterations taken to solve the problem.

PRESOLVE_TIME
> indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME
> indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

Examples: OPTMILP Procedure

This section contains examples that illustrate the options and syntax of PROC OPTMILP. Example 13.1 demonstrates a model contained in an MPS-format SAS data set and finds an optimal solution by using PROC OPTMILP. Example 13.2 illustrates the use of standard MPS files in PROC OPTMILP. Example 13.3 demonstrates how to warm start PROC OPTMILP. More detailed examples of mixed integer linear programs, along with example SAS code, are given in Chapter 8.

Example 13.1: Simple Integer Linear Program

This example illustrates a model in an MPS-format SAS data set. This data set is passed to PROC OPTMILP, and a solution is found.

Consider a scenario where you have a container with a set of limiting attributes (volume V and weight W) and a set I of items that you want to pack. Each item type i has a certain value p_i, a volume v_i, and a weight w_i. You must choose at most four items of each type so that the total value is maximized and all the chosen items fit into the container. Let x_i be the number of items of type i to be included in the container. This model can be formulated as the following integer linear program:

$$
\begin{array}{llll}
\max & \sum_{i \in I} p_i x_i & & \\
\text{s.t.} & \sum_{i \in I} v_i x_i & \leq \quad V & \text{(volume\_con)} \\
& \sum_{i \in I} w_i x_i & \leq \quad W & \text{(weight\_con)} \\
& x_i & \leq \quad 4 & \forall i \in I \\
& x_i \in \mathbb{Z}^+ & & \forall i \in I
\end{array}
$$

Constraint (volume_con) enforces the volume capacity limit, while constraint (weight_con) enforces the weight capacity limit. An instance of this problem can be saved in an MPS-format SAS data set by using the following code:

```
data ex1data;
    input field1 $ field2 $ field3 $ field4 field5 $ field6;
    datalines;
NAME          .               ex1data        .       .               .
ROWS          .               .              .       .               .
MAX           z               .              .       .               .
L             volume_con      .              .       .               .
L             weight_con      .              .       .               .
COLUMNS       .               .              .       .               .
.             .MRK0           'MARKER'        .       'INTORG'        .
.             x[1]            z               1       volume_con      10
.             x[1]            weight_con      12      .               .
.             x[2]            z               2       volume_con      300
.             x[2]            weight_con      15      .               .
.             x[3]            z               3       volume_con      250
.             x[3]            weight_con      72      .               .
.             x[4]            z               4       volume_con      610
.             x[4]            weight_con      100     .               .
.             x[5]            z               5       volume_con      500
.             x[5]            weight_con      223     .               .
.             x[6]            z               6       volume_con      120
.             x[6]            weight_con      16      .               .
.             x[7]            z               7       volume_con      45
.             x[7]            weight_con      73      .               .
.             x[8]            z               8       volume_con      100
.             x[8]            weight_con      12      .               .
.             x[9]            z               9       volume_con      200
.             x[9]            weight_con      200     .               .
.             x[10]           z               10      volume_con      61
.             x[10]           weight_con      110     .               .
.             .MRK1           'MARKER'        .       'INTEND'        .
RHS           .               .              .       .               .
.             .RHS.           volume_con      1000    .               .
.             .RHS.           weight_con      500     .               .
BOUNDS        .               .              .       .               .
UP            .BOUNDS.        x[1]            4       .               .
UP            .BOUNDS.        x[2]            4       .               .
UP            .BOUNDS.        x[3]            4       .               .
UP            .BOUNDS.        x[4]            4       .               .
UP            .BOUNDS.        x[5]            4       .               .
UP            .BOUNDS.        x[6]            4       .               .
UP            .BOUNDS.        x[7]            4       .               .
UP            .BOUNDS.        x[8]            4       .               .
UP            .BOUNDS.        x[9]            4       .               .
UP            .BOUNDS.        x[10]           4       .               .
ENDATA        .               .              .       .               .
;
```

In the COLUMNS section of this data set, the name of the objective is z, and the objective coefficients p_i appear in field4. The coefficients v_i of (volume_con) appear in field6. The coefficients w_i of (weight_con) appear in field4. In the RHS section, the bounds V and W appear in field4.

This problem can be solved by using the following statements to call the OPTMILP procedure:

```
proc optmilp data=ex1data primalout=ex1soln;
run;
```

The progress of the solver is shown in Output 13.1.1.

Output 13.1.1 Simple Integer Linear Program PROC OPTMILP Log

```
NOTE: The problem ex1data has 10 variables (0 binary, 10 integer, 0 free, 0
      fixed).
NOTE: The problem has 2 constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 20 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 2 variables and 0 constraints.
NOTE: The MILP presolver removed 4 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 8 variables, 2 constraints, and 16 constraint
      coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node   Active    Sols    BestInteger      BestBound      Gap     Time
             0        1       3      85.0000000    158.0000000   46.20%        0
             0        1       3      85.0000000     88.0955497    3.51%        0
             0        1       3      85.0000000     88.0955497    3.51%        0
             0        1       3      85.0000000     88.0955497    3.51%        0
NOTE: The MILP presolver is applied again.
             0        1       3      85.0000000     88.0955497    3.51%        0
             0        1       3      85.0000000     88.0626822    3.48%        0
             0        1       3      85.0000000     87.8820655    3.28%        0
             0        1       3      85.0000000     87.8539763    3.25%        0
             0        1       3      85.0000000     87.7208690    3.10%        0
             0        1       3      85.0000000     87.7180302    3.10%        0
             0        1       3      85.0000000     87.7133502    3.09%        0
             0        1       3      85.0000000     87.7128245    3.09%        0
NOTE: The MILP solver added 2 cuts with 8 cut coefficients at the root.
             3        1       4      86.0000000     87.3854749    1.59%        0
             4        0       5      87.0000000     87.0000000    0.00%        0
NOTE: Optimal.
NOTE: Objective = 87.
NOTE: The data set WORK.EX1SOLN has 10 observations and 8 variables.
```

The data set ex1soln is shown in Output 13.1.2.

Output 13.1.2 Simple Integer Linear Program Solution

Example 1 Solution Data

Objective Function ID	RHS ID	Variable Name	Variable Type	Objective Coefficient	Lower Bound	Upper Bound	Variable Value
z	.RHS.	x[1]	I	1	0	4	0
z	.RHS.	x[2]	I	2	0	4	0
z	.RHS.	x[3]	I	3	0	4	0
z	.RHS.	x[4]	I	4	0	4	0
z	.RHS.	x[5]	I	5	0	4	0
z	.RHS.	x[6]	I	6	0	4	3
z	.RHS.	x[7]	I	7	0	4	1
z	.RHS.	x[8]	I	8	0	4	4
z	.RHS.	x[9]	I	9	0	4	0
z	.RHS.	x[10]	I	10	0	4	3

The optimal solution is $x_6 = 3, x_7 = 1, x_8 = 4$, and $x_{10} = 3$, with a total value of 87. From this solution, you can compute the total volume used, which is 988 ($\leq V = 1000$); the total weight used is 499 ($\leq W = 500$). The problem summary and solution summary are shown in Output 13.1.3.

Output 13.1.3 Simple Integer Linear Program Summary

The OPTMILP Procedure

Problem Summary	
Problem Name	ex1data
Objective Sense	Maximization
Objective Function	z
RHS	.RHS.
Number of Variables	10
Bounded Above	0
Bounded Below	0
Bounded Above and Below	10
Free	0
Fixed	0
Binary	0
Integer	10
Number of Constraints	2
LE (<=)	2
EQ (=)	0
GE (>=)	0
Range	0
Constraint Coefficients	20

Output 13.1.3 *continued*

Performance Information	
Execution Mode	Single-Machine
Number of Threads	4

Solution Summary	
Solver	MILP
Algorithm	Branch and Cut
Objective Function	z
Solution Status	Optimal
Objective Value	87
Relative Gap	0
Absolute Gap	0
Primal Infeasibility	0
Bound Infeasibility	0
Integer Infeasibility	0
Best Bound	87
Nodes	5
Iterations	160
Presolve Time	0.00
Solution Time	0.01

Example 13.2: MIPLIB Benchmark Instance

The following example illustrates the conversion of a standard MPS-format file into an MPS-format SAS data set. The problem is re-solved several times, each time by using a different control option. For such a small example, it is necessary to disable cuts and heuristics in order to see the computational savings gained by using other options. For larger or more complex examples, the benefits of using the various control options are more pronounced.

The standard set of MILP benchmark cases is called MIPLIB (Bixby et al. 1998, Achterberg, Koch, and Martin 2003) and can be found at http://miplib.zib.de/. The following statement uses the %MPS2SASD macro to convert an example from MIPLIB to a SAS data set:

```
%mps2sasd(mpsfile="bell3a.mps", outdata=mpsdata);
```

The problem can then be solved using PROC OPTMILP on the data set created by the conversion:

```
proc optmilp data=mpsdata allcuts=none heuristics=none logfreq=10000;
run;
```

The resulting log is shown in Output 13.2.1.

Output 13.2.1 MIPLIB PROC OPTMILP Log

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 44 variables and 58 constraints.
NOTE: The MILP presolver removed 142 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 89 variables, 65 constraints, and 205
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node   Active    Sols    BestInteger      BestBound      Gap     Time
             0        1       0              .         869515        .        0
           891      136       1         907340         873167    3.91%        0
          1814      128       3         898595         874502    2.76%        0
          1874      126       4         878651         874502    0.47%        0
          5124     1341       5         878430         875369    0.35%        0
         10000     1517       5         878430         876097    0.27%        0
         14778        0       5         878430         878430    0.00%        0
NOTE: Optimal.
NOTE: Objective = 878430.316.
```

Suppose you do not have a bound for the solution. If there is an objective value that, even if it is not optimal, satisfies your requirements, then you can save time by using the TARGET= option. The following PROC OPTMILP call solves the problem with a target value of 880,000:

```
proc optmilp data=mpsdata allcuts=none heuristics=none logfreq=5000
              target=880000;
run;
```

The relevant results from this run are displayed in Output 13.2.2. In this case, there is a decrease in CPU time, but the objective value has increased.

Output 13.2.2 MIPLIB PROC OPTMILP Log with TARGET= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 44 variables and 58 constraints.
NOTE: The MILP presolver removed 142 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 89 variables, 65 constraints, and 205
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active   Sols   BestInteger      BestBound      Gap    Time
             0       1      0             .          869515       .       0
           891     136      1        907340          873167    3.91%      0
          1814     128      3        898595          874502    2.76%      0
          1874     126      4        878651          874502    0.47%      0
NOTE: Target reached.
NOTE: Objective of the best integer solution found = 878651.068.
```

When the objective value of a solution is within a certain relative gap of the optimal objective value, the procedure stops. The acceptable relative gap can be changed using the RELOBJGAP= option, as demonstrated in the following example:

```
proc optmilp data=mpsdata allcuts=none heuristics=none relobjgap=0.01;
run;
```

The relevant results from this run are displayed in Output 13.2.3. In this case, since the specified RELOBJ-GAP= value is larger than the default value, the number of nodes and the CPU time have decreased from their values in the original run. Note that these savings are exchanged for an increase in the objective value of the solution.

Output 13.2.3 MIPLIB PROC OPTMILP Log with RELOBJGAP= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 44 variables and 58 constraints.
NOTE: The MILP presolver removed 142 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 89 variables, 65 constraints, and 205
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	0	.	869515	.	0
100	82	0	.	872669	.	0
200	159	0	.	872669	.	0
300	235	0	.	872908	.	0
400	314	0	.	872908	.	0
500	392	0	.	872935	.	0
600	463	0	.	873094	.	0
700	534	0	.	873094	.	0
800	46	0	.	873167	.	0
891	136	1	907340	873167	3.91%	0
900	144	1	907340	873167	3.91%	0
1000	56	2	907340	873441	3.88%	0
1100	132	2	907340	874032	3.81%	0
1200	210	2	907340	874058	3.81%	0
1300	289	2	907340	874165	3.80%	0
1400	368	2	907340	874218	3.79%	0
1500	443	2	907340	874314	3.78%	0
1600	517	2	907340	874435	3.76%	0
1700	17	2	907340	874502	3.76%	0
1800	117	2	907340	874502	3.76%	0
1814	128	3	898595	874502	2.76%	0
1874	126	4	878651	874502	0.47%	0

```
NOTE: Optimal within relative gap.
NOTE: Objective = 878651.068.
```

The MAXTIME= option enables you to accept the best solution produced by PROC OPTMILP in a specified amount of time. The following example illustrates the use of the MAXTIME= option:

```
proc optmilp data=mpsdata allcuts=none heuristics=none maxtime=0.1;
run;
```

The relevant results from this run are displayed in Output 13.2.4. Once again, a reduction in solution time is traded for an increase in objective value.

Output 13.2.4 MIPLIB PROC OPTMILP Log with MAXTIME= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 44 variables and 58 constraints.
NOTE: The MILP presolver removed 142 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 89 variables, 65 constraints, and 205
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	0	.	869515	.	0
100	82	0	.	872669	.	0
200	159	0	.	872669	.	0
300	235	0	.	872908	.	0
400	314	0	.	872908	.	0
500	392	0	.	872935	.	0
600	463	0	.	873094	.	0
700	534	0	.	873094	.	0
800	46	0	.	873167	.	0
891	136	1	907340	873167	3.91%	0
900	144	1	907340	873167	3.91%	0
1000	56	2	907340	873441	3.88%	0
1100	132	2	907340	874032	3.81%	0
1200	210	2	907340	874058	3.81%	0

```
NOTE: Real time limit reached.
NOTE: Objective of the best integer solution found = 907340.09.
```

The MAXNODES= option enables you to limit the number of nodes generated by PROC OPTMILP. The following example illustrates the use of the MAXNODES= option:

```
proc optmilp data=mpsdata allcuts=none heuristics=none maxnodes=1000;
run;
```

The relevant results from this run are displayed in Output 13.2.5. PROC OPTMILP displays the best objective value of all the solutions produced.

Output 13.2.5 MIPLIB PROC OPTMILP Log with MAXNODES= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0
      fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 44 variables and 58 constraints.
NOTE: The MILP presolver removed 142 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 89 variables, 65 constraints, and 205
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	0	.	869515	.	0
100	82	0	.	872669	.	0
200	159	0	.	872669	.	0
300	235	0	.	872908	.	0
400	314	0	.	872908	.	0
500	392	0	.	872935	.	0
600	463	0	.	873094	.	0
700	534	0	.	873094	.	0
800	46	0	.	873167	.	0
891	136	1	907340	873167	3.91%	0
900	144	1	907340	873167	3.91%	0
999	55	2	907340	873441	3.88%	0

```
NOTE: Node limit reached.
NOTE: Objective of the best integer solution found = 907340.09.
```

Example 13.3: Facility Location

This advanced example demonstrates how to warm start PROC OPTMILP by using the PRIMALIN= option. The model is constructed in PROC OPTMODEL and saved in an MPS-format SAS data set for use in PROC OPTMILP. This problem can also be solved from within PROC OPTMODEL; see Chapter 8 for details.

Consider the classical facility location problem. Given a set *L* of customer locations and a set *F* of candidate facility sites, you must decide on which sites to build facilities and assign coverage of customer demand to these sites so as to minimize cost. All customer demand d_i must be satisfied, and each facility has a demand capacity limit *C*. The total cost is the sum of the distances c_{ij} between facility *j* and its assigned customer *i*, plus a fixed charge f_j for building a facility at site *j*. Let $y_j = 1$ represent choosing site *j* to build a facility, and 0 otherwise. Also, let $x_{ij} = 1$ represent the assignment of customer *i* to facility *j*, and 0 otherwise. This model can be formulated as the following integer linear program:

$$\min \ \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_j y_j$$

$$
\begin{array}{llll}
\text{s.t.} & \displaystyle\sum_{j \in F} x_{ij} & = \ 1 & \forall i \in L & \text{(assign\_def)} \\[2ex]
& x_{ij} & \le \ y_j & \forall i \in L, j \in F & \text{(link)} \\[1ex]
& \displaystyle\sum_{i \in L} d_i x_{ij} & \le \ C y_j & \forall j \in F & \text{(capacity)} \\[2ex]
& x_{ij} \in \{0, 1\} & & \forall i \in L, j \in F & \\[1ex]
& y_j \in \{0, 1\} & & \forall j \in F &
\end{array}
$$

Constraint (assign_def) ensures that each customer is assigned to exactly one site. Constraint (link) forces a facility to be built if any customer has been assigned to that facility. Finally, constraint (capacity) enforces the capacity limit at each site.

Consider also a variation of this same problem where there is no cost for building a facility. This problem is typically easier to solve than the original problem. For this variant, let the objective be

$$\min \ \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij}$$

First, construct a random instance of this problem by using the following DATA steps:

```
%let NumCustomers  = 50;
%let NumSites      = 10;
%let SiteCapacity  = 35;
%let MaxDemand     = 10;
%let xmax          = 200;
%let ymax          = 100;
%let seed          = 938;

/* generate random customer locations */
data cdata(drop=i);
   length name $8;
   do i = 1 to &NumCustomers;
      name = compress('C'||put(i,best.));
      x = ranuni(&seed) * &xmax;
      y = ranuni(&seed) * &ymax;
      demand = ranuni(&seed) * &MaxDemand;
      output;
   end;
run;

/* generate random site locations and fixed charge */
data sdata(drop=i);
   length name $8;
   do i = 1 to &NumSites;
      name = compress('SITE'||put(i,best.));
      x = ranuni(&seed) * &xmax;
```

```
            y = ranuni(&seed) * &ymax;
            fixed_charge = 30 * (abs(&xmax/2-x) + abs(&ymax/2-y));
            output;
        end;
    run;
```

The following PROC OPTMODEL statements generate the model and define both variants of the cost function:

```
proc optmodel;
    set <str> CUSTOMERS;
    set <str> SITES init {};
    /* x and y coordinates of CUSTOMERS and SITES */
    num x {CUSTOMERS union SITES};
    num y {CUSTOMERS union SITES};
    num demand {CUSTOMERS};
    num fixed_charge {SITES};
    /* distance from customer i to site j */
    num dist {i in CUSTOMERS, j in SITES}
        = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);
    read data cdata into CUSTOMERS=[name] x y demand;
    read data sdata into SITES=[name] x y fixed_charge;
    var Assign {CUSTOMERS, SITES} binary;
    var Build {SITES} binary;
    /* each customer assigned to exactly one site */
    con assign_def {i in CUSTOMERS}:
        sum {j in SITES} Assign[i,j] = 1;
    /* if customer i assigned to site j, then facility must be */
    /* built at j                                              */
    con link {i in CUSTOMERS, j in SITES}:
        Assign[i,j] <= Build[j];
    /* each site can handle at most &SiteCapacity demand */
    con capacity {j in SITES}:
        sum {i in CUSTOMERS} demand[i] * Assign[i,j]
        <= &SiteCapacity * Build[j];
    min CostNoFixedCharge
        = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j];
    save mps nofcdata;
    min CostFixedCharge
        = CostNoFixedCharge
          + sum {j in SITES} fixed_charge[j] * Build[j];
    save mps fcdata;
quit;
```

First solve the problem for the model with no fixed charge by using the following statements. The first PROC SQL call populates the macro variables varcostNo. This macro variable displays the objective value when the results are plotted. The second PROC SQL call generates a data set that is used to plot the results. The information printed in the log by PROC OPTMILP is displayed in Output 13.3.1.

```
proc optmilp data=nofcdata primalout=nofcout;
run;
proc sql noprint;
   select put(sum(_objcoef_ * _value_),6.1) into :varcostNo
   from nofcout;
quit;

proc sql;
   create table CostNoFixedCharge_Data as
   select
      scan(p._var_,2,'[],') as customer,
      scan(p._var_,3,'[],') as site,
      c.x as xi, c.y as yi, s.x as xj, s.y as yj
   from
      cdata as c,
      sdata as s,
      nofcout(where=(substr(_var_,1,6)='Assign' and
                    round(_value_) = 1)) as p
   where calculated customer = c.name and calculated site = s.name;
quit;
```

Output 13.3.1 PROC OPTMILP Log for Facility Location with No Fixed Charges

```
NOTE: The problem nofcdata has 510 variables (510 binary, 0 integer, 0 free, 0
      fixed).
NOTE: The problem has 560 constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 10 variables and 500 constraints.
NOTE: The MILP presolver removed 1010 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 500 variables, 60 constraints, and 1000
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
         Node  Active   Sols    BestInteger      BestBound      Gap     Time
            0       1      2      972.1737321              0    972.2        0
            0       1      2      972.1737321    961.2403449    1.14%        0
            0       1      2      972.1737321    961.2403449    1.14%        0
            0       1      2      972.1737321    961.2403449    1.14%        0
NOTE: The MILP presolver is applied again.
            0       1      4      966.4832160    962.9120804    0.37%        0
            0       1      5      966.4832160    966.4832160    0.00%        0
            0       0      5      966.4832160    966.4832160    0.00%        0
NOTE: The MILP solver added 0 cuts with 0 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 966.48321599.
NOTE: The data set WORK.NOFCOUT has 510 observations and 8 variables.
```

Next, solve the fixed-charge model by using the following statements. Note that the solution to the model with no fixed charge is feasible for the fixed-charge model and should provide a good starting point for PROC OPTMILP. The PRIMALIN= option provides an incumbent solution ("warm start"). The two PROC SQL calls perform the same functions as in the case with no fixed charges. The results from this approach are shown in Output 13.3.2.

```
proc optmilp data=fcdata primalin=nofcout primalout=fcout;
run;
proc sql noprint;
   select put(sum(_objcoef_ * _value_), 6.1) into :varcost
   from fcout(where=(substr(_var_,1,6)='Assign'));
   select put(sum(_objcoef_ * _value_), 5.1) into :fixcost
   from fcout(where=(substr(_var_,1,5)='Build'));
   select put(sum(_objcoef_ * _value_), 6.1) into :totalcost
   from fcout;
quit;
proc sql;
   create table CostFixedCharge_Data as
   select
      scan(p._var_,2,'[],') as customer,
      scan(p._var_,3,'[],') as site,
      c.x as xi, c.y as yi, s.x as xj, s.y as yj
   from
      cdata as c,
      sdata as s,
      fcout(where=(substr(_var_,1,6)='Assign' and
                   round(_value_) = 1)) as p
   where calculated customer = c.name and calculated site = s.name;
quit;
```

Output 13.3.2 PROC OPTMILP Log for Facility Location with Fixed Charges, Using Warm Start

```
NOTE: The problem fcdata has 510 variables (510 binary, 0 integer, 0 free, 0
      fixed).
NOTE: The problem has 560 constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 510 variables, 560 constraints, and 2010
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node   Active    Sols    BestInteger        BestBound       Gap      Time
             0        1       3   16070.0150023                0     16070         0
             0        1       3   16070.0150023    9946.2514269    61.57%         0
             0        1       3   16070.0150023    9962.4849932    61.31%         0
             0        1       3   16070.0150023    9971.5893075    61.16%         0
             0        1       3   16070.0150023    9974.5588580    61.11%         0
             0        1       3   16070.0150023    9978.1322942    61.05%         0
             0        1       3   16070.0150023    9978.3312183    61.05%         0
             0        1       3   16070.0150023    9980.4930282    61.01%         0
             0        1       3   16070.0150023    9981.2701907    61.00%         0
             0        1       4   16009.9864088    9981.2701907    60.40%         0
NOTE: The MILP solver added 20 cuts with 631 cut coefficients at the root.
            13        8       5   11115.0946280   10027.3840001    10.85%         0
            92       38       6   10986.9615361   10298.3588908     6.69%         0
           101       33       7   10966.7191607   10405.7274389     5.39%         0
           143       14       8   10959.4361909   10789.5675920     1.57%         0
           276       48       9   10950.0345545   10945.3793911     0.04%         0
           348       55      10   10948.4603465   10946.5600201     0.02%         0
           377       40      10   10948.4603465   10947.6598816     0.01%         0
NOTE: Optimal within relative gap.
NOTE: Objective = 10948.460346.
NOTE: The data set WORK.FCOUT has 510 observations and 8 variables.
```

The following two SAS programs produce a plot of the solutions for both variants of the model, using data sets produced by PROC SQL from the PRIMALOUT= data sets produced by PROC OPTMILP.

NOTE: Execution of this code requires SAS/GRAPH software.

```
title1 "Facility Location Problem";
title2 "TotalCost = &varcostNo (Variable = &varcostNo, Fixed = 0)";
data csdata;
   set cdata(rename=(y=cy)) sdata(rename=(y=sy));
run;
/* create Annotate data set to draw line between customer and */
/* assigned site                                              */
%annomac;
data anno(drop=xi yi xj yj);
   %SYSTEM(2, 2, 2);
set CostNoFixedCharge_Data(keep=xi yi xj yj);
   %LINE(xi, yi, xj, yj, *, 1, 1);
run;
proc gplot data=csdata anno=anno;
   axis1 label=none order=(0 to &xmax by 10);
   axis2 label=none order=(0 to &ymax by 10);
   symbol1 value=dot interpol=none
      pointlabel=("#name" nodropcollisions height=0.7) cv=black;
   symbol2 value=diamond interpol=none
      pointlabel=("#name" nodropcollisions color=blue height=0.7) cv=blue;
   plot cy*x sy*x / overlay haxis=axis1 vaxis=axis2;
run;
quit;
```

The output from the first program appears in Output 13.3.3.

Output 13.3.3 Solution Plot for Facility Location with No Fixed Charges
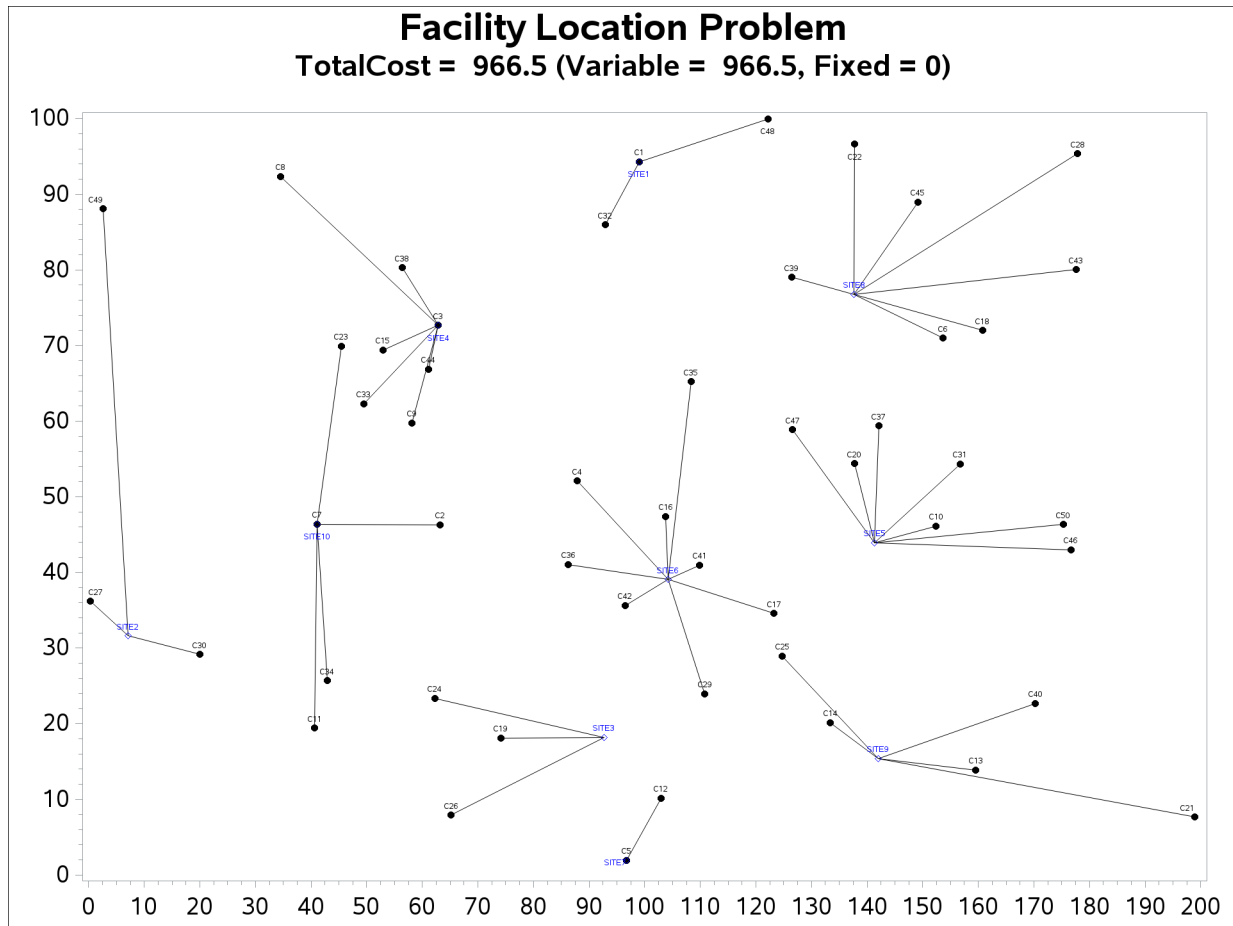

```
title1 "Facility Location Problem";
title2 "TotalCost = &totalcost (Variable = &varcost, Fixed = &fixcost)";
/* create Annotate data set to draw line between customer and */
/* assigned site                                              */
data anno(drop=xi yi xj yj);
   %SYSTEM(2, 2, 2);
   set CostFixedCharge_Data(keep=xi yi xj yj);
   %LINE(xi, yi, xj, yj, *, 1, 1);
run;
proc gplot data=csdata anno=anno;
   axis1 label=none order=(0 to &xmax by 10);
   axis2 label=none order=(0 to &ymax by 10);
   symbol1 value=dot interpol=none
      pointlabel=("#name" nodropcollisions height=0.7) cv=black;
   symbol2 value=diamond interpol=none
      pointlabel=("#name" nodropcollisions color=blue height=0.7) cv=blue;
   plot cy*x sy*x / overlay haxis=axis1 vaxis=axis2;
run;
quit;
```

The output from the second program appears in Output 13.3.4.

Output 13.3.4 Solution Plot for Facility Location with Fixed Charges
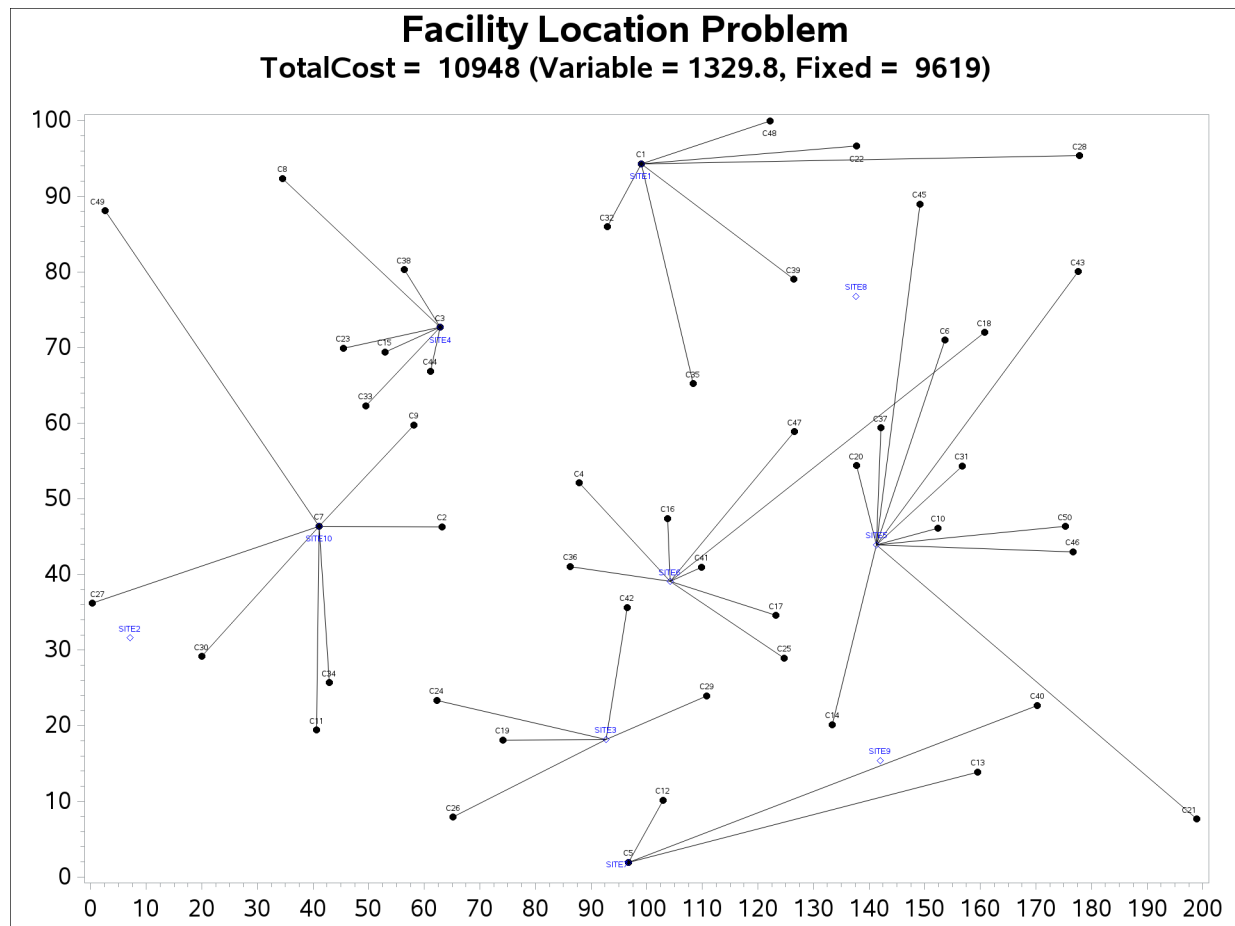
The economic tradeoff for the fixed-charge model forces you to build fewer sites and push more demand to each site.

Example 13.4: Scheduling

This example is intended for users who prefer to use the SAS DATA step, PROC SQL, and similar programming methods to prepare data for input to SAS/OR optimization procedures. SAS/OR users who prefer to use the algebraic modeling capabilities of PROC OPTMODEL to specify optimization models should consult Example 8.1 in Chapter 8, "The Mixed Integer Linear Programming Solver," for a discussion of the same business problem in a PROC OPTMODEL context.

Scheduling is an application area where techniques in model generation can be valuable. Problems that involve scheduling are often solved with integer programming and are similar to assignment problems. In this example, you have eight one-hour time slots in each of five days. You have to assign four people to these time slots so that each slot is covered every day. You allow the people to specify preference data for each slot on each day. In addition, there are constraints that must be satisfied:

- Each person has some slots for which they are unavailable.

- Each person must have either slot 4 or 5 off for lunch.

- Each person can work only two time slots in a row.

- Each person can work only a specified number of hours in the week.

To formulate this problem, let i denote person, j denote time slot, and k denote day. Then, let $x_{ijk} = 1$ if person i is assigned to time slot j on day k, and 0 otherwise; let p_{ijk} denote the preference of person i for slot j on day k; and let h_i denote the number of hours in a week that person i will work. Then, you get

$$
\begin{array}{lll}
\max & \sum_{ijk} p_{ijk} x_{ijk} & \\
\text{subject to} & \sum_i x_{ijk} = 1 & \text{for all } j \text{ and } k \\
& x_{i4k} + x_{i5k} \leq 1 & \text{for all } i \text{ and } k \\
& x_{i,\ell,k} + x_{i,\ell+1,k} + x_{i,\ell+2,k} \leq 2 & \text{for all } i \text{ and } k, \text{ and } \ell = 1, \ldots, 6 \\
& \sum_{jk} x_{ijk} \leq h_i & \text{for all } i \\
& x_{ijk} = 0 \text{ or } 1 & \text{for all } i \text{ and } k \text{ such that } p_{ijk} > 0, \\
& & \qquad \text{otherwise } x_{ijk} = 0
\end{array}
$$

To solve this problem, create a data set that has the hours and preference data for each individual, time slot, and day. A 10 represents the most desirable time slot, and a 1 represents the least desirable time slot. In addition, a 0 indicates that the time slot is not available.

```
data raw;
   input name $ hour slot mon tue wed thu fri;
   datalines;
marc   20    1     10 10 10 10 10
marc   20    2      9  9  9  9  9
marc   20    3      8  8  8  8  8
marc   20    4      1  1  1  1  1
marc   20    5      1  1  1  1  1
marc   20    6      1  1  1  1  1
marc   20    7      1  1  1  1  1
marc   20    8      1  1  1  1  1
mike   20    1     10  9  8  7  6
mike   20    2     10  9  8  7  6
mike   20    3     10  9  8  7  6
mike   20    4     10  3  3  3  3
mike   20    5      1  1  1  1  1
mike   20    6      1  2  3  4  5
mike   20    7      1  2  3  4  5
mike   20    8      1  2  3  4  5
bill   20    1     10 10 10 10 10
bill   20    2      9  9  9  9  9
bill   20    3      8  8  8  8  8
bill   20    4      0  0  0  0  0
bill   20    5      1  1  1  1  1
bill   20    6      1  1  1  1  1
bill   20    7      1  1  1  1  1
bill   20    8      1  1  1  1  1
bob    20    1     10  9  8  7  6
bob    20    2     10  9  8  7  6
bob    20    3     10  9  8  7  6
```

```
bob    20    4    10   3   3   3   3
bob    20    5     1   1   1   1   1
bob    20    6     1   2   3   4   5
bob    20    7     1   2   3   4   5
bob    20    8     1   2   3   4   5
;
```

These data are read by the following DATA step, and an integer program is built to solve the problem. The model is saved in the data set named MODEL, which is constructed in the following steps:

1. The objective function is built using the data saved in the RAW data set.

2. The constraints that ensure that no one works during a time slot during which they are unavailable are built.

3. The constraints that require a person to be working in each time slot are built.

4. The constraints that allow each person time for lunch are added.

5. The constraints that restrict people to only two consecutive hours are added.

6. The constraints that limit the time that any one person works in a week are added.

7. The constraints that allow a person to be assigned only to a time slot for which he is available are added.

The statements to build each of these constraints follow the formulation closely.

```
data model;
   array workweek{5} mon tue wed thu fri;
   array hours{4} hours1 hours2 hours3 hours4;
   retain hours1-hours4;

   set raw end=eof;

   length _row_ $ 8 _col_ $ 8 _type_ $ 8;
   keep _type_ _col_ _row_ _coef_;

   if      name='marc' then i=1;
   else if name='mike' then i=2;
   else if name='bill' then i=3;
   else if name='bob'  then i=4;

   hours{i}=hour;

   /* build the objective function */

   do k=1 to 5;
      _col_='x'||put(i,1.)||put(slot,1.)||put(k,1.);

      _row_='object';
      _coef_=workweek{k} * 1000;
      output;
   end;

   /* build the rest of the model */
```

```
/* cannot work during unavailable slots */
do k=1 to 5;
   if workweek{k}=0 then do;
      _row_='off'||put(i,1.)||put(slot,1.)||put(k,1.);
      _type_='eq';
      _col_='_RHS_';
      _coef_=0;
    output;
      _col_='x'||put(i,1.)||put(slot,1.)||put(k,1.);
     _coef_=1;
     _type_=' ';
    output;
   end;
end;

if eof then do;
   _coef_=.;
   _col_=' ';

   /* every hour 1 person working */
   do j=1 to 8;
      do k=1 to 5;
         _row_='work'||put(j,1.)||put(k,1.);
         _type_='eq';
         _col_='_RHS_';
         _coef_=1;
         output;
         _coef_=1;
         _type_=' ';
         do i=1 to 4;
             _col_='x'||put(i,1.)||put(j,1.)||put(k,1.);
            output;
         end;
      end;
   end;

   /* each person has a lunch */
   do i=1 to 4;
      do k=1 to 5;
         _row_='lunch'||put(i,1.)||put(k,1.);
         _type_='le';
         _col_='_RHS_';
         _coef_=1;
         output;
         _coef_=1;
         _type_=' ';
         _col_='x'||put(i,1.)||'4'||put(k,1.);
         output;
         _col_='x'||put(i,1.)||'5'||put(k,1.);
         output;
      end;
   end;
```

```
        /* work at most 2 slots in a row */
        do i=1 to 4;
           do k=1 to 5;
              do l=1 to 6;
              _row_='seq'||put(i,1.)||put(k,1.)||put(l,1.);
              _type_='le';
              _col_='_RHS_';
              _coef_=2;
               output;
              _coef_=1;
              _type_=' ';
                 do j=0 to 2;
                    _col_='x'||put(i,1.)||put(l+j,1.)||put(k,1.);
                    output;
                 end;
              end;
           end;
        end;

        /* work at most n hours in a week */
        do i=1 to 4;
           _row_='capacit'||put(i,1.);
           _type_='le';
           _col_='_RHS_';
           _coef_=hours{i};
           output;
           _coef_=1;
           _type_=' ';
           do j=1 to 8;
              do k=1 to 5;
                 _col_='x'||put(i,1.)||put(j,1.)||put(k,1.);
                 output;
              end;
           end;
        end;
     end;
  run;
```

Next, this SAS data set is converted to an MPS-format SAS data set by establishing the structure of the MPS format and through very minor conversions of the data.

```
  /* the following code transforms the above sparse data set */
  /* into an MPS-format data set                             */

  /* generate the header of the MPS-format data set */
  data mps0;
     format field1 field2 field3 $10.;
     format field4 10.;
     format field5 $10.;
     format field6 10.;
     field1 = 'NAME';
     field2 = '        ';
     field3 = 'PROBLEM';
     field4 = .;
```

```
      field5 = '        ';
      field6 = .;
      output;
      field1 = 'ROWS';
      field3 = '';
      output;
      field1 = 'MAX';
      field2 = 'object';
      field3 = '';
      output;
run;

/* generate rows */
proc sql;
   create table mps1 as
      select _type_ as field1, _row_ as field2 from model
         where _row_ eq 'object' and _type_ ne '' union
      select 'E' as field1, _row_ as field2 from model
         where _type_ eq 'eq' union
      select 'L' as field1, _row_ as field2 from model
         where _type_ eq 'le' union
      select 'G' as field1, _row_ as field2 from model
         where _type_ eq 'ge';
quit;

/* indicate start of columns section and declare type of all */
/* variables as integer                                      */
data mps2;
   format field1 field2 field3  $10.;
   format field4 10.;
   format field5 $10.;
   format field6 10.;
   field1 = 'COLUMNS';
   field2 = '          ';
   field3 = '          ';
   field4 = .;
   field5 = '          ';
   field6 = .;
   output;
   field1 = '          ';
   field2 = '.MARK0';
   field3 = "'MARKER'";
   field4 = .;
   field5 = "'INTORG'";
   field6 = .;
   output;
run;

/* generate columns */
data mps3;
   set model;
   format field1 field2 field3 $10.;
   format field4 10.;
```

```
      format field5 $10.;
      format field6 10.;
      keep field1-field6;
      field1 = '         ';
      field2 = _col_;
      field3 = _row_;
      field4 = _coef_;
      field5 = '         ';
      field6 = .;
      if field2 ne '_RHS_'  then do;
         output;
      end;
   run;

   /* sort columns by variable names */
   proc sort data=mps3;
      by field2;
   run;

   /* indicate the end of the columns section */
   data mps4;
      format field1 field2 field3  $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      field1 = '           ';
      field2 = '.MARK1';
      field3 = "'MARKER'";
      field4 = .;
      field5 = "'INTEND'";
      field6 = .;
      output;
   run;

   /* indicate the start of the RHS section */
   data mps5;
      format field1 field2 field3  $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      field1 = 'RHS';
   run;

   /* generate RHS entries */
   data mps6;
      set model;
      format field1 field2 field3 $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      keep field1-field6;
      field1 = '        ';
      field2 = _col_;
      field3 = _row_;
```

```
      field4 = _coef_;
      field5 = '         ';
      field6 = .;
      if field2 eq '_RHS_'  then do;
         output;
      end;
   run;

   /* denote the end of the MPS-format data set */
   data mps7;
      format field1 field2 field3  $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      field1 = 'ENDATA';
   run;

   /* merge all sections of the MPS-format data set */
   data mps;
      format field1 field2 field3 $10.;
      format field4 10.;
      format field5 $10.;
      format field6 10.;
      set mps0 mps1 mps2 mps3 mps4 mps5 mps6 mps7;
   run;
```

The model is solved using the OPTMILP procedure. The option PRIMALOUT=SOLUTION causes PROC OPTMILP to save the primal solution in the data set named SOLUTION.

```
   /* solve the binary program */
   proc optmilp data=mps
      printlevel=0 loglevel=0
      primalout=solution maxtime=1000;
   run;
```

The following DATA step takes the solution data set SOLUTION and generates a report data set named REPORT. It restores the original interpretation (person, shift, day) of the variable names xijk so that a more meaningful report can be written. Then PROC TABULATE is used to display a schedule that shows how the eight time slots are covered for the week.

```
/* report the solution */
title 'Reported Solution';

data report;
   set solution;
   keep name slot mon tue wed thu fri;
   if substr(_var_,1,1)='x' then do;
      if _value_>0 then do;
         n=substr(_var_,2,1);
         slot=substr(_var_,3,1);
         d=substr(_var_,4,1);
         if      n='1' then name='marc';
         else if n='2' then name='mike';
         else if n='3' then name='bill';
         else               name='bob';
         if      d='1' then mon=1;
         else if d='2' then tue=1;
         else if d='3' then wed=1;
         else if d='4' then thu=1;
         else               fri=1;
         output;
      end;
   end;
run;

proc format;
   value xfmt 1='  xxx   ';
run;

proc tabulate data=report;
   class name slot;
   var mon--fri;
   table (slot * name), (mon tue wed thu fri)*sum=' '*f=xfmt.
         /misstext=' ';
run;
```

Output 13.4.1 from PROC TABULATE summarizes the schedule. Notice that the constraint that requires a person to be assigned to each possible time slot on each day is satisfied.

Output 13.4.1 A Scheduling Problem

Reported Solution

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

Recall that PROC OPTMILP puts a character string in the macro variable _OROPTMILP_ that describes the characteristics of the solution on termination. This string can be parsed using macro functions, and the information obtained can be used in report writing. The variable can be written to the log with the following command:

```
%put &_OROPTMILP_;
```

This command produces the output shown in Output 13.4.2.

Output 13.4.2 _OROPTMILP_ Macro Variable

```
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=211000 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0 BEST_BOUND=211000 NODES=1 ITERATIONS=64
PRESOLVE_TIME=0.00 SOLUTION_TIME=0.01
```

From this output you learn, for example, that at termination the solution is integer-optimal and has an objective value of 211,000.

References

Achterberg, T., Koch, T., and Martin, A. (2003). "MIPLIB 2003." http://miplib.zib.de/.

Achterberg, T., Koch, T., and Martin, A. (2005). "Branching Rules Revisited." *Operations Research Letters* 33:42–54.

Andersen, E. D., and Andersen, K. D. (1995). "Presolving in Linear Programming." *Mathematical Programming* 71:221–245.

Atamturk, A. (2004). "Sequence Independent Lifting for Mixed-Integer Programming." *Operations Research* 52:487–490.

Bixby, R. E., Ceria, S., McZeal, C. M., and Savelsbergh, M. W. P. (1998). "An Updated Mixed Integer Programming Library: MIPLIB 3.0." *Optima* 58:12–15.

Dantzig, G. B., Fulkerson, R., and Johnson, S. M. (1954). "Solution of a Large-Scale Traveling Salesman Problem." *Operations Research* 2:393–410.

Gondzio, J. (1997). "Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method." *INFORMS Journal on Computing* 9:73–91.

Land, A. H., and Doig, A. G. (1960). "An Automatic Method for Solving Discrete Programming Problems." *Econometrica* 28:497–520.

Linderoth, J. T., and Savelsbergh, M. W. P. (1998). "A Computational Study of Search Strategies for Mixed Integer Programming." *INFORMS Journal on Computing* 11:173–187.

Marchand, H., Martin, A., Weismantel, R., and Wolsey, L. (1999). "Cutting Planes in Integer and Mixed Integer Programming." DP 9953, CORE, Université Catholique de Louvain.

Ostrowski, J. (2008). *Symmetry in Integer Programming*. Ph.D. diss., Lehigh University.

Savelsbergh, M. W. P. (1994). "Preprocessing and Probing Techniques for Mixed Integer Programming Problems." *ORSA Journal on Computing* 6:445–454.

Subject Index

Syntax Index