Chapter 9
The Mixed Integer Linear Programming Solver

Overview: MILP Solver

The OPTMODEL procedure provides a framework for specifying and solving mixed integer linear programs (MILPs). A standard mixed integer linear program has the formulation

\[
\begin{align*}
\min & \quad c^T x \\
\text{subject to} & \quad Ax \begin{cases} \geq, =, \leq \end{cases} b \\
& \quad 1 \leq x \leq u \\
& \quad x_i \in \mathbb{Z} \quad \forall i \in S
\end{align*}
\] (MILP)
where

\[\begin{align*}
    & x \in \mathbb{R}^n & \text{is the vector of structural variables} \\
    & A \in \mathbb{R}^{m \times n} & \text{is the matrix of technological coefficients} \\
    & c \in \mathbb{R}^n & \text{is the vector of objective function coefficients} \\
    & b \in \mathbb{R}^m & \text{is the vector of constraints’ right-hand sides (RHS)} \\
    & l \in \mathbb{R}^n & \text{is the vector of lower bounds on variables} \\
    & u \in \mathbb{R}^n & \text{is the vector of upper bounds on variables} \\
    & S & \text{is a nonempty subset of the set } \{1 \ldots n\} \text{ of indices}
\end{align*}\]

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

The MILP solver provides various control options and solution strategies. In particular, you can enable, disable, or set levels for the advanced techniques previously mentioned. It is also possible to input an incumbent solution; see the section “Warm Start Option” on page 366 for details.

---

**Getting Started: MILP Solver**

The following example illustrates how you can use the OPTMODEL procedure to solve mixed integer linear programs. For more examples, see the section “Examples: MILP Solver” on page 386. Suppose you want to solve the following problem:

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

You can use the following statements to call the OPTMODEL procedure for solving mixed integer linear programs:

```optmodel
proc optmodel;
    var x{1..3} >= 0 integer;

    min f = 2*x[1] - 3*x[2] - 4*x[3];


    solve with milp / presolver = automatic heuristics = automatic;
    print x;
quit;
```
The **PRESOLVER=** and **HEURISTICS=** options specify the levels for presolving and applying heuristics, respectively. In this example, each option is set to its default value, **AUTOMATIC**, meaning that the solver automatically determines the appropriate levels for presolve and heuristics.

The optimal value of \( x \) is shown in Figure 9.1.

**Figure 9.1** Solution Output

The **OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
</tr>
<tr>
<td>2 1</td>
</tr>
<tr>
<td>3 1</td>
</tr>
</tbody>
</table>

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

```plaintext
%put &_OROPTMODEL;
```

This statement produces the output shown in Figure 9.2.

**Figure 9.2** Macro Output

| STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=-7 RELATIVE_GAP=0 ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0 INTEGER_INFEASIBILITY=0 BEST_BOUND=-7 NODES=1 SOLUTIONS_FOUND=2 ITERATIONS=3 PRESOLVE_TIME=0.00 SOLUTION_TIME=0.01 |

---

**Syntax: MILP Solver**

The following statement is available in the **OPTMODEL procedure**:

```plaintext
SOLVE WITH MILP / options;
```

---

**Functional Summary**

Table 9.1 summarizes the options available for the SOLVE WITH MILP statement, classified by function.

<table>
<thead>
<tr>
<th><strong>Table 9.1</strong> Options for the MILP Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Presolve Option</td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
</tr>
<tr>
<td>Warm Start Option</td>
</tr>
<tr>
<td>Specifies the input primal solution</td>
</tr>
<tr>
<td>(warm start)</td>
</tr>
<tr>
<td>Control Options</td>
</tr>
<tr>
<td>Specifies the stopping criterion</td>
</tr>
<tr>
<td>based on absolute objective gap</td>
</tr>
</tbody>
</table>
### Table 9.1 (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the cutoff value for node removal</td>
<td>CUTOFF=</td>
</tr>
<tr>
<td>Emphasizes feasibility or optimality</td>
<td>EMPHASIS=</td>
</tr>
<tr>
<td>Specifies the maximum violation on variables and constraints</td>
<td>FEASTOL=</td>
</tr>
<tr>
<td>Specifies the maximum allowed difference between an integer variable’s value and an integer</td>
<td>INTTOL=</td>
</tr>
<tr>
<td>Specifies the frequency of printing the node log</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the detail of solution progress printed in log</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum number of nodes to be processed</td>
<td>MAXNODES=</td>
</tr>
<tr>
<td>Specifies the maximum number of solutions to return from the pool</td>
<td>MAXPOOLSOLS=</td>
</tr>
<tr>
<td>Specifies the maximum number of solutions to be found</td>
<td>MAXSOLS=</td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the tolerance used in determining the optimality of nodes in the branch-and-bound tree</td>
<td>OPTTOL=</td>
</tr>
<tr>
<td>Specifies the probing level</td>
<td>PROBE=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on relative objective gap</td>
<td>RELOBJGAP=</td>
</tr>
<tr>
<td>Enables the use of scaling on the problem matrix</td>
<td>SCALE=</td>
</tr>
<tr>
<td>Specifies the initial seed for the random number generator</td>
<td>SEED=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on target objective value</td>
<td>TARGET=</td>
</tr>
<tr>
<td>Specifies whether time units are CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
</tbody>
</table>

**Heuristics Option**

<table>
<thead>
<tr>
<th>Specifies the primal heuristics level</th>
<th>HEURISTICS=</th>
</tr>
</thead>
</table>

**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 to perform on each variable in the strong branching variable selection strategy | STRONGITER=   |
| Specifies the number of candidates for the strong branching variable selection strategy | STRONGLEN=   |
| Specifies the level of symmetry detection                                 | SYMMETRY=     |
| Specifies the rule for selecting the 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                                              | CUTFGOMORY=   |
| 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=       |
**MILP Solver Options**

This section describes the options that are recognized by the MILP solver in PROC OPTMODEL. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the MILP solver is explicitly specified using a WITH clause. For example, the following line could appear in PROC OPTMODEL statements:

```plaintext
solve with milp / allcuts=aggressive maxnodes=10000 primalin;
```

**Presolve Option**

**PRESOLVER=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE**

specifies the level of presolve processing. You can specify the following values:

- **AUTOMATIC** applies the default level of presolve processing.
- **NONE** disables the presolver.
- **BASIC** performs minimal presolve processing.
- **MODERATE** applies a higher level of presolve processing.
- **AGGRESSIVE** applies the highest level of presolve processing.

By default, PRESOLVER=AUTOMATIC.
Warm Start Option

**PRIMALIN**

enables you to input a starting solution in PROC OPTMODEL before invoking the MILP solver. Adding the PRIMALIN option to the SOLVE statement requests that the MILP solver use the current variable values as a starting solution (warm start). If the MILP solver finds that the input solution is feasible, then the input solution provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution is not feasible, the MILP solver tries to repair it. It is possible to set a variable value to the missing value ‘.’ to mark a variable for repair. When it is difficult to find a good integer feasible solution for a problem, warm start can reduce solution time significantly.

**NOTE:** If the MILP solver produces a feasible solution, the variable values from that run can be used as the warm start solution for a subsequent run. If the warm start solution is not feasible for the subsequent run, the solver automatically tries to repair it.

Control Options

**ABSOBJGAP=** *number*

**ABSOLUTEOBJECTIVEGAP=** *number*

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

**CUTOFF=** *number*

cuts off any nodes in a minimization (maximization) problem that have an objective value at or above (below) *number*. The value of *number* can be any number; the default value is the largest (smallest) number that can be represented by a double.

**EMPHASIS=** **BALANCE | OPTIMAL | FEASIBLE**

specifies a search emphasis *string* as listed below.

- **BALANCE** performs a balanced search.
- **OPTIMAL** emphasizes optimality over feasibility.
- **FEASIBLE** emphasizes feasibility over optimality.

By default, EMPHASIS=BALANCE.

**FEASTOL=** *number*

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

If the MILP solver fails to find a feasible solution within this tolerance but does find a solution that has some violation, then the solver stops with a solution status of OPTIMAL_COND (see the section “Macro Variable _OROPTMODEL_ ” on page 383).
**INTTOL=number**

**INTEGERTOLERANCE=number**

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

If the best solution that the solver finds has an integer infeasibility larger than the value of *number*, then the solver stops with a solution status of OPTIMAL_COND (see the section “Macro Variable _OROPTMODEL_” on page 383).

**LOGFREQ=k**

**PRINTFREQ=k**

Prints information in the node log every *k* seconds, where *k* is any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} − 1$. If *k*=0, then the node log is disabled. If *k* is positive, then the root node processing information is printed and, if possible, an entry is made every *k* seconds. An entry is also made each time a better integer solution is found.

By default, LOGFREQ=5.

**LOGLEVEL=NONE | BASIC | MODERATE | AGGRESSIVE**

Controls the amount of information displayed in the SAS log by the MILP solver. You can specify the following values:

- **NONE** turns off all solver-related messages in the SAS log.
- **BASIC** displays a solver summary after stopping.
- **MODERATE** prints a solver summary and a node log by using the interval specified in the LOGFREQ= option.
- **AGGRESSIVE** prints a detailed solver summary and a node log by using the interval specified in the LOGFREQ= option.

By default, LOGLEVEL=MODERATE.

**MAXNODES=number**

Specifies the maximum number of branch-and-bound nodes to be processed, where *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} − 1$. If you run the MILP solver in concurrent mode (CONCURRENT=TRUE), then the solver stops as soon as *number* is reached on any machine. If you run the MILP solver in distributed mode (DISTRIBUTED=TRUE), then the solver periodically checks and stops as soon as the total number of nodes that are processed by all grid nodes exceeds *number*. The default value of *number* is $2^{31} − 1$.

**MAXPOOLSOLS=number**

Specifies the number of solutions to return from the MILP solution pool, where *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} − 1$. Only feasible and unique solutions are returned, and they are returned in order of objective value, with the best solution first. The number of solutions that are found is reported in both the solution summary and the _OROPTMODEL_ macro variable. By default, MAXPOOLSOLS=1.
**MAXSOLS=** number

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

**MAXTIME=** t

specifies an upper limit of t units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop because of the amount of time elapsed. If concurrent or distributed mode of the solver is enabled (CONCURRENT=TRUE or DISTRIBUTED=TRUE), then the solver stops as soon as t is reached on any machine. The value of t can be any positive number; the default value is the largest number that can be represented by a double.

**OPTTOL=** number

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

**PROBE=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies a probing strategy. You can specify the following values:

- **AUTOMATIC** uses the probing strategy determined by the MILP solver.
- **NONE** disables probing.
- **MODERATE** uses the probing moderately.
- **AGGRESSIVE** uses probing aggressively.

By default, PROBE=AUTOMATIC.

**RELOBJGAP=** number

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

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

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

**SCALE=AUTOMATIC | NONE**

indicates whether to scale the problem matrix. You can specify the following values:

- **AUTOMATIC** scales the matrix as determined by the MILP solver.
- **NONE** disables scaling.

By default, SCALE=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 $2^{31} - 1$. By default, SEED=100.
**TARGET=number**

specifies a stopping criterion for a minimization or maximization problem. If the best integer objective is better than or equal to number, the solver stops. The value of number can be any number; the default value is the largest (in magnitude) negative number (for a minimization problem) or the largest (in magnitude) positive number (for a maximization problem) that can be represented by a double.

**TIMETYPE=CPU | REAL**

specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. You can specify the following values:

- **CPU** specifies that units are in CPU time.
- **REAL** specifies that units are in real time.

The “Optimization Statistics” table, an output of PROC OPTMODEL if you specify PRINTLEVEL=2 in the PROC OPTMODEL statement, also includes the same time units for Presolver Time and Solver Time. The other times (such as Problem Generation Time) in the “Optimization Statistics” table are also in the same units.

By default, TIMETYPE=REAL.

**Heuristics Option**

**HEURISTICS=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE**

controls the level of primal heuristics applied by the MILP solver. This level determines how frequently primal heuristics are applied during the branch-and-bound tree search. It also affects the maximum number of iterations allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. You can specify the following values:

- **AUTOMATIC** applies the default level of heuristics, similar to MODERATE.
- **NONE** disables all primal heuristics. This value does not disable the heuristics that repair an infeasible input solution that is specified by using the PRIMALIN option.
- **BASIC** applies basic primal heuristics at low frequency.
- **MODERATE** applies most primal heuristics at moderate frequency.
- **AGGRESSIVE** applies all primal heuristics at high frequency.

By default, HEURISTICS=AUTOMATIC. For more information about primal heuristics, see the section “Primal Heuristics” on page 380.

**Search Options**

**CONFLICTSEARCH=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of conflict search performed by the MILP solver. A conflict search finds clauses resulting from infeasible subproblems that arise in the search tree. You can specify the following values:
Chapter 9: The Mixed Integer Linear Programming Solver

**AUTOMATIC** performs conflict search based on a strategy determined by the MILP solver.

**NONE** disables conflict search.

**MODERATE** performs a moderate conflict search.

**AGGRESSIVE** performs an aggressive conflict search.

By default, **CONFLICTSEARCH=** AUTOMATIC.

**NODESEL=AUTOMATIC | BESTBOUND | BESTESTIMATE | DEPTH** specifies the node selection strategy. You can specify the following values:

- **AUTOMATIC** uses automatic node selection.
- **BESTBOUND** chooses the node with the best relaxed objective (best-bound-first strategy).
- **BESTESTIMATE** chooses the node with the best estimate of the integer objective value (best-estimate-first strategy).
- **DEPTH** chooses the most recently created node (depth-first strategy).

By default, **NODESEL=AUTOMATIC**. For more information about node selection, see the section “Node Selection” on page 376.

**PRIORITY= TRUE | FALSE** indicates whether to use specified branching priorities for integer variables. You can specify the following values:

- **TRUE** uses priorities when they exist.
- **FALSE** ignores variable priorities.

By default, **PRIORITY=TRUE**. For more information, see the section “Branching Priorities” on page 378.

**RESTARTS=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE** specifies the strategy for restarting the processing of the root node. You can specify the following values:

- **AUTOMATIC** uses a restarting strategy determined by the MILP solver.
- **NONE** disables restarting.
- **BASIC** uses a basic restarting strategy.
- **MODERATE** uses a moderate restarting strategy.
- **AGGRESSIVE** uses an aggressive restarting strategy.

By default, **RESTARTS=AUTOMATIC**.

**STRONGITER=** number | AUTOMATIC

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

**SYMMETRY=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE**
specifies the level of symmetry detection. Symmetry detection identifies groups of equivalent decision variables and uses this information to solve the problem more efficiently. You can specify the following values:

- **AUTOMATIC** performs symmetry detection based on a strategy that is determined by MILP solver.
- **NONE** disables symmetry detection.
- **BASIC** performs a basic symmetry detection.
- **MODERATE** performs a moderate symmetry detection.
- **AGGRESSIVE** performs an aggressive symmetry detection.

By default, SYMMETRY=AUTOMATIC. For more information about symmetry detection, see (Ostrowski 2008).

**VARSEL=AUTOMATIC | MAXINFEAS | MININFEAS | PSEUDO | STRONG**
specifies the rule for selecting the branching variable. You can specify the following values:

- **AUTOMATIC** uses automatic branching variable selection.
- **MAXINFEAS** chooses the variable with maximum infeasibility.
- **MININFEAS** chooses the variable with minimum infeasibility.
- **PSEUDO** chooses a branching variable based on pseudocost.
- **STRONG** uses a strong branching variable selection strategy.

By default, VARSEL=AUTOMATIC. For more information about variable selection, see the section “Variable Selection” on page 377.

**Cut Options**

Table 9.2 describes the *string* values for the cut options in the OPTMODEL procedure.

<table>
<thead>
<tr>
<th><em>string</em></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMATIC</td>
<td>Generates cutting planes based on a strategy determined by the MILP solver</td>
</tr>
<tr>
<td>NONE</td>
<td>Disables generation of cutting planes</td>
</tr>
<tr>
<td>MODERATE</td>
<td>Uses a moderate cut strategy</td>
</tr>
<tr>
<td>AGGRESSIVE</td>
<td>Uses an aggressive cut strategy</td>
</tr>
</tbody>
</table>
You can specify the `CUTSTRATEGY=` option to set the overall aggressiveness of the cut generation in the MILP solver. Alternatively, you can use the `ALLCUTS=` option to set all cut types to the same level. You can override the `ALLCUTS=` value by using the options that correspond to particular cut types. For example, if you want the MILP solver to generate only Gomory cuts, specify `ALLCUTS=NONE` and `CUTGOMORY=AUTOMATIC`. If you want to generate all cuts aggressively but generate no lift-and-project cuts, set `ALLCUTS=AGGRESSIVE` and `CUTLAP=NONE`.

`ALLCUTS=AUTOMATIC | NONE | MODERATE | AGGRESSIVE`

provides a shorthand way of setting all the cuts-related options in one setting. In other words, `ALLCUTS=string` is equivalent to setting each of the individual cuts parameters to the same value `string`. Thus, `ALLCUTS=AUTOMATIC` has the effect of setting `CUTCLIQUE=AUTOMATIC`, `CUTFLOWCOVER=AUTOMATIC`, `CUTFLOWPATH=AUTOMATIC`, ..., `CUTMULTICOMMODITY=AUTOMATIC`, and `CUTZEROHALF=AUTOMATIC`. Table 9.2 lists the values that can be assigned to `option`. In addition, you can override levels for individual cuts with the `CUTCLIQUE=`, `CUTFLOWCOVER=`, `CUTFLOWPATH=`, `CUTGOMORY=`, `CUTGUB=`, `CUTIMPLIED=`, `CUTKNAPSACK=`, `CUTLAP=`, `CUTMILIFTED=`, `CUTMIR=`, `CUTMULTICOMMODITY=`, and `CUTZEROHALF=` options. If the `ALLCUTS=` option is not specified, then all the cuts-related options are either at their individually specified values (if the corresponding option is specified) or at their default values (if that option is not specified).

`CUTCLIQUE=AUTOMATIC | NONE | MODERATE | AGGRESSIVE`

specifies the level of clique cuts that are generated by the MILP solver. Table 9.2 describes the possible values. This option overrides the `ALLCUTS=` option. By default, `CUTCLIQUE=AUTOMATIC`.

`CUTFLOWCOVER=AUTOMATIC | NONE | MODERATE | AGGRESSIVE`

specifies the level of flow cover cuts that are generated by the MILP solver. Table 9.2 describes the possible values. The option overrides the `ALLCUTS=` option. By default, `CUTFLOWCOVER=AUTOMATIC`.

`CUTFLOWPATH=AUTOMATIC | NONE | MODERATE | AGGRESSIVE`

specifies the level of flow path cuts that are generated by the MILP solver. Table 9.2 describes the possible values. This option overrides the `ALLCUTS=` option. By default, `CUTFLOWPATH=AUTOMATIC`.

`CUTGOMORY=AUTOMATIC | NONE | MODERATE | AGGRESSIVE`

specifies the level of Gomory cuts that are generated by the MILP solver. Table 9.2 describes the possible values. This option overrides the `ALLCUTS=` option. By default, `CUTGOMORY=AUTOMATIC`.

`CUTGUB=AUTOMATIC | NONE | MODERATE | AGGRESSIVE`

specifies the level of generalized upper bound (GUB) cover cuts that are generated by the MILP solver. Table 9.2 describes the possible values. This option overrides the `ALLCUTS=` option. By default, `CUTGUB=AUTOMATIC`.

`CUTIMPLIED=AUTOMATIC | NONE | MODERATE | AGGRESSIVE`

specifies the level of implied bound cuts that are generated by the MILP solver. Table 9.2 describes the possible values. This option overrides the `ALLCUTS=` option. By default, `CUTIMPLIED=AUTOMATIC`.
CUTKNAPSACK=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of knapsack cover cuts that are generated by the MILP solver. Table 9.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTKNAPSACK=AUTOMATIC.

CUTLAP=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of lift-and-project (LAP) cuts that are generated by the MILP solver. Table 9.2 describes the possible values that can be assigned to option. This option overrides the ALLCUTS= option. By default, CUTLAP=NONE.

CUTMILIFTED=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of mixed lifted 0-1 cuts that are generated by the MILP solver. Table 9.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTMILIFTED=AUTOMATIC.

CUTMIR=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of mixed integer rounding (MIR) cuts that are generated by the MILP solver. Table 9.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTMIR=AUTOMATIC.

CUTMULTICOMMODITY=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of multicommodity network flow cuts that are generated by the MILP solver. Table 9.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTMULTICOMMODITY=AUTOMATIC.

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

CUTSTRATEGY=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
CUTS=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the overall aggressiveness of the cut generation in the solver. By default, CUTSTRATEGY=AUTOMATIC. Setting a nondefault value adjusts a number of cut parameters such that the cut generation is none, moderate, or aggressive compared to the default value.

CUTZEROHALF=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of zero-half cuts that are generated by the MILP solver. Table 9.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTZEROHALF=AUTOMATIC.

Decomposition Algorithm Options
The following options are available for the decomposition algorithm in the MILP solver. For information about the decomposition algorithm, see Chapter 16, “The Decomposition Algorithm.”

DECOMP=(options)
enables the decomposition algorithm and specifies overall control options for the algorithm. For more information about this option, see Chapter 16, “The Decomposition Algorithm.”
DECOMPMASTER=(options)
specifies options for the master problem. For more information about this option, see Chapter 16, “The Decomposition Algorithm.”

DECOMPMASTERIP=(options)
specifies options for the (restricted) master problem solved as a MILP with the current set of columns in an effort to obtain an integer feasible solution. For more information about this option, see Chapter 16, “The Decomposition Algorithm.”

DECOMPSUBPROB=(options)
specifies option for the subproblem. For more information about this option, see Chapter 16, “The Decomposition Algorithm.”

Parallel Options

CONCURRENT=TRUE | FALSE
specifies whether to run the MILP solver in concurrent mode. You can specify the following values:

TRUE runs the MILP solver in concurrent mode. The solver automatically runs the same problem on each grid node with different MILP solver options.
FALSE turns off concurrent mode for the MILP solver.

By default, CONCURRENT=FALSE.

DETERMINISTIC=TRUE | FALSE
specifies whether to run the MILP solver deterministically when distributed or concurrent mode is enabled. The DETERMINISTIC= option is valid only when DISTRIBUTED=TRUE or CONCURRENT=TRUE. You can specify the following values:

TRUE runs the MILP solver in deterministic distributed or concurrent mode. The solver always returns the same solution for subsequent runs on the same grid.
FALSE runs the MILP solver in nondeterministic distributed or concurrent mode.

By default, DETERMINISTIC=FALSE when you specify CONCURRENT=TRUE, and DETERMINISTIC=TRUE when you specify DISTRIBUTED=TRUE.

DISTRIBUTED=TRUE | FALSE
specifies whether to run the MILP solver in distributed mode. For more information about this option as it relates to the standard MILP algorithm, see the section “Parallel Processing” on page 380. For more information about this option as it relates to the decomposition algorithm, see Chapter 16, “The Decomposition Algorithm.” You can specify the following values:

TRUE runs the MILP solver in distributed mode.
FALSE turns off distributed mode for the MILP solver.

By default, DISTRIBUTED=FALSE.
**Details: MILP Solver**

**NTHREADS=number**

specifies the maximum number of threads for the MILP solver to use for multithreaded processing, where *number* can be any integer between 1 and 256, inclusive. The branch-and-cut algorithm can take advantage of multicore machines and can potentially run faster when *number* is greater than 1. The default is the value of the NTHREADS= option in PROC OPTMODEL.

---

**Branch-and-Bound Algorithm**

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

The branch-and-bound algorithm solves a mixed integer linear program by dividing the search space and generating a sequence of subproblems. The search space of a mixed integer linear program can be represented by a tree. Each node in the tree is identified with a subproblem derived from previous subproblems on the path that leads to the root of the tree. The subproblem (MILP⁰) associated with the root is identical to the original problem, which is called (MILP), given in the section “Overview: MILP Solver” on page 361.

The linear programming relaxation (LP⁰) of (MILP⁰) can be written as

\[
\begin{align*}
\min & \quad c^T x \\
\text{subject to} & \quad Ax \{\geq, =, \leq\} b \\
& \quad 1 \leq x \leq u
\end{align*}
\]

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 (LP⁰), which is usually obtained by using the dual simplex algorithm. If \( \bar{x}^0_i \) is an integer for all \( i \in S \), then \( \bar{x}^0 \) is an optimal solution to (MILP). Suppose that for some \( i \in S \), \( \bar{x}^0_i \) is nonintegral. In that case the algorithm defines two new subproblems (MILP¹) and (MILP²), descendants of the parent subproblem (MILP⁰). The subproblem (MILP¹) is identical to (MILP⁰) except for the additional constraint

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

and the subproblem (MILP²) is identical to (MILP⁰) except for the additional constraint

\[
x_i \geq \lceil x^0_i \rceil
\]

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 \( i \in S \)), 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 376 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 nondeterministic polynomial-time hard (NP-hard). Roughly speaking, this means that the effort required to solve a mixed integer linear program grows exponentially with the size of the problem. For example, a problem with 10 binary variables can generate in the worst case \( 2^{10} = 1,024 \) nodes in the branch-and-bound tree. A problem with 20 binary variables can generate in the worst case \( 2^{20} = 1,048,576 \) nodes in the branch-and-bound tree. Although it is unlikely that the branch-and-bound algorithm has to generate every single possible node, the need to explore even a small fraction of the potential number of nodes for a large problem can be resource-intensive.

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

**Controlling the Branch-and-Bound Algorithm**

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

\[ f_i(k) = \tilde{x}_i^k - \lfloor \tilde{x}_i^k \rfloor \]

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

**Node Selection**

The NODESEL= option specifies the strategy used to select the next active node. The valid keywords for this option are AUTOMATIC, BESTBOUND, BESTESTIMATE, and DEPTH. The following list describes the strategy associated with each keyword:
AUTOMATIC enables the MILP solver to choose the best node selection strategy based on problem characteristics and search progress. This is the default setting.

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

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

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

**Variable Selection**

The VARSEL= option specifies the strategy used to select the next branching variable. The valid keywords for this option are AUTOMATIC, MAXINFEAS, MININFEAS, PSEUDO, and STRONG. The following list describes the action taken in each case when \( \bar{x}^k \), a relaxed optimal solution of (MILP\(^k\)), is used to define two active subproblems. In the following list, “INTTOL” refers to the value assigned using the INTTOL= option. For details about the INTTOL= option, see the section “Control Options” on page 366.

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

MAXINFEAS chooses as the branching variable the variable \( x_i \) such that \( i \) maximizes

\[
\{ \min \{ f_i(k), 1 - f_i(k) \} \mid i \in S \text{ and } \\
\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL} \}
\]

MININFEAS chooses as the branching variable the variable \( x_i \) such that \( i \) minimizes

\[
\{ \min \{ f_i(k), 1 - f_i(k) \} \mid i \in S \text{ and } \\
\text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL} \}
\]

PSEUDO chooses as the branching variable the variable \( x_i \) such that \( i \) maximizes the weighted up and down pseudocosts. Pseudocost branching attempts to branch on significant variables first, quickly improving lower bounds. Pseudocost branching estimates significance based on historical information; however, this approach might not be accurate for future search.
STRONG chooses as the branching variable the variable $x_i$ such that $i$ maximizes the estimated improvement in the objective value. Strong branching first generates a list of candidates, then branches on each candidate and records the improvement in the objective value. The candidate with the largest improvement is chosen as the branching variable. Strong branching can be effective for combinatorial problems, but it is usually computationally expensive.

**Branching Priorities**

In some cases, it is possible to speed up the branch-and-bound algorithm by branching on variables in a specific order. You can accomplish this in PROC OPTMODEL by attaching branching priorities to the integer variables in your model by using the `.priority` suffix. More information about this suffix is available in the section “Integer Variable Suffixes” on page 139 in Chapter 5. For an example in which branching priorities are used, see Example 9.3.

**Presolve and Probing**

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

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

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

**Cutting Planes**

The feasible region of every linear program forms a polyhedron. Every polyhedron in $n$-space can be written as a finite number of half-spaces (equivalently, inequalities). In the notation used in this chapter, this polyhedron is defined by the set $Q = \{x \in \mathbb{R}^n \mid Ax \leq b, l \leq x \leq u\}$. After you add the restriction that some variables must be integral, the set of feasible solutions, $F = \{x \in Q \mid x_i \in \mathbb{Z} \ \forall i \in S\}$, no longer forms a polyhedron.

The convex hull of a set $X$ is the minimal convex set that contains $X$. In solving a mixed integer linear program, in order to take advantage of LP-based algorithms you want to find the convex hull, $\text{conv}(F)$, of $F$. If you can find $\text{conv}(F)$ 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 375, the branch-and-bound algorithm begins by solving the linear programming relaxation over the polyhedron \( Q \). Clearly, \( Q \) contains the convex hull of the feasible region of the original integer program; that is, \( \text{conv}(F) \subseteq Q \).

Cutting plane techniques are used to tighten the linear relaxation to better approximate \( \text{conv}(F) \). Assume you are given a solution \( \bar{x} \) to some intermediate linear relaxation during the branch-and-bound algorithm. A cut, or valid inequality \((\pi x \leq \pi^0)\), is some half-space with the following characteristics:

- The half-space contains \( \text{conv}(F) \); that is, every integer feasible solution is feasible for the cut \((\pi x \leq \pi^0, \forall x \in F)\).
- The half-space does not contain the current solution \( \bar{x} \); that is, \( \bar{x} \) is not feasible for the cut \((\pi \bar{x} > \pi^0)\).

Cutting planes were first made popular by Dantzig, Fulkerson, and Johnson (1954) in their work on the traveling salesman problem. The two major classifications of cutting planes are generic cuts and structured cuts. Generic cuts are based solely on algebraic arguments and can be applied to any relaxation of any integer program. Structured cuts are specific to certain structures that can be found in some relaxations of the mixed integer linear program. These structures are automatically discovered during the cut initialization phase of the MILP solver. Table 9.3 lists the various types of cutting planes that are built into the MILP solver. Included in each type are algorithms for numerous variations based on different relaxations and lifting techniques. For a survey of cutting plane techniques for mixed integer programming, see Marchand et al. (1999). For a survey of lifting techniques, see Atamturk (2004).

**Table 9.3** Cutting Planes in the MILP Solver

<table>
<thead>
<tr>
<th>Generic Cutting Planes</th>
<th>Structured Cutting Planes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gomory mixed integer</td>
<td>Cliques</td>
</tr>
<tr>
<td>Lift-and-project</td>
<td>Flow cover</td>
</tr>
<tr>
<td>Mixed integer rounding</td>
<td>Flow path</td>
</tr>
<tr>
<td>Mixed lifted 0-1</td>
<td>Generalized upper bound cover</td>
</tr>
<tr>
<td>Zero-half</td>
<td>Implied bound</td>
</tr>
<tr>
<td></td>
<td>Knapsack cover</td>
</tr>
<tr>
<td></td>
<td>Multicommodity network flow</td>
</tr>
</tbody>
</table>

You can set levels for individual cuts by using the `CUTCLIQUE=`, `CUTFLOWCOVER=`, `CUTFLOWPATH=`, `CUTGOMORY=`, `CUTGUB=`, `CUTIMPLIED=`, `CUTKNAPSACK=`, `CUTLAP=`, `CUTMILIFTED=`, `CUTMIR=`, `CUTMULTICOMMODITY=`, and `CUTZEROHALF=` options. The valid levels for these options are listed in Table 9.2.

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

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

The default settings of the MILP solver have been tuned to work well for most instances. However, problem-specific expertise might suggest adjusting one or more of the strategies. These options give you that flexibility.
Chapter 9: The Mixed Integer Linear Programming Solver

Primal Heuristics

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

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

- finding a good upper bound early in the tree search (this can lead to earlier fathoming, resulting in fewer subproblems to be processed)
- locating a reasonably good feasible solution when that is sufficient (sometimes a reasonably good feasible solution is the best the solver can produce within certain time or resource limits)
- providing upper bounds for some bound-tightening techniques

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

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

Parallel Processing

You can run the branch-and-cut algorithm in the following modes:

- In single-machine mode, the computation is executed on a single computer by multiple threads. The number of threads is specified in the NTHREADS= option.
- In distributed mode, the branch-and-cut algorithm is executed on a cluster of nodes. You disable deterministic distributed mode by specifying DETERMINISTIC=FALSE in the MILP solver options. By default, the MILP solver is executed deterministically.
- In concurrent mode, the computation is executed on a cluster of nodes by solving the same problem across those nodes under different MILP solver settings. By default, the MILP solver is executed nondeterministically.

You can run the decomposition algorithm in either single-machine or distributed mode. In distributed mode, the computation is executed on multiple computing nodes in a distributed computing environment.
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 that are found so far. You can use the MAXPOOLSOLS= option to obtain more than one solution. Note that the number printed in the node log can be different from the number of found solutions that is reported in the solution summary and in the _OROPTMODEL_ macro variable.

BestInteger indicates the best upper bound (assuming minimization) that is found so far.

BestBound indicates the best lower bound (assuming minimization) that is 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 or CPU time.

If CONCURRENT=TRUE, then the following information about the status of the branch-and-bound algorithm is printed in the node log:

Node indicates the total number of nodes that are processed by all concurrent solves.

Active indicates the minimum number of active nodes among all concurrent solves.

Sols indicates the number of feasible solutions that are found by all concurrent solves.

BestInteger indicates the best upper bound (assuming minimization) that is found by all concurrent solves.

BestBound indicates the best lower bound (assuming minimization) that is found by all concurrent solves.

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 or CPU time.

If CONCURRENT=TRUE, the node log displays the progress of the overall grid. If DETERMINISTIC=TRUE, the solver tries to return a reproducible solution. Therefore, there might be discrepancies between the node log and the final solution that is reported by the deterministic concurrent MILP solver.

The LOGFREQ= option can be used to control the amount of information that is printed in the node log. By default, the root node processing information is printed and, if possible, an entry is made every five seconds. 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 9.3 shows a sample node log.
Figure 9.3  Sample Node Log

NOTE: Problem generation will use 16 threads.
NOTE: The problem has 510 variables (0 free, 0 fixed).
NOTE: The problem uses 1 implicit variables.
NOTE: The problem has 510 binary and 0 integer variables.
NOTE: The problem has 560 linear constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The initial MILP heuristics are applied.
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 16 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>0</td>
<td>24087</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19197.7909681</td>
<td>25.47%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19206.4233772</td>
<td>25.41%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19210.6103687</td>
<td>25.38%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19217.2333020</td>
<td>25.34%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>21700.2572819</td>
<td>19217.2333020</td>
<td>12.92%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>21700.2572819</td>
<td>19219.6945880</td>
<td>12.91%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>21700.2572819</td>
<td>19224.1234459</td>
<td>12.88%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>21693.6723572</td>
<td>19224.1234459</td>
<td>12.84%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>21693.6723572</td>
<td>19227.8702614</td>
<td>12.82%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP solver added 20 cuts with 741 cut coefficients at the root.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>77</td>
<td>51</td>
<td>9</td>
<td>21686.3282331</td>
<td>19292.0971396</td>
<td>12.41%</td>
<td>0</td>
</tr>
<tr>
<td>122</td>
<td>60</td>
<td>11</td>
<td>21554.0701956</td>
<td>19314.7035626</td>
<td>11.59%</td>
<td>0</td>
</tr>
<tr>
<td>353</td>
<td>14</td>
<td>12</td>
<td>21548.6159565</td>
<td>20326.7015711</td>
<td>6.01%</td>
<td>0</td>
</tr>
<tr>
<td>385</td>
<td>0</td>
<td>12</td>
<td>21548.6159565</td>
<td>21546.8592630</td>
<td>0.01%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Optimal within relative gap.
NOTE: Objective = 21548.615957.

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of $10^9$) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMODEL procedure causes the ODS table ProblemStatistics to be generated when the MILP solver is called. This table provides basic data magnitude information that enables you to improve the formulation of your models.
The example output in Figure 9.4 demonstrates the contents of the ODS table ProblemStatistics.

**Figure 9.4** ODS Table ProblemStatistics

<table>
<thead>
<tr>
<th>Obs</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of Constraint Matrix Nonzeros</td>
<td>8</td>
<td>8.00000</td>
</tr>
<tr>
<td>2</td>
<td>Maximum Constraint Matrix Coefficient</td>
<td>3</td>
<td>3.00000</td>
</tr>
<tr>
<td>3</td>
<td>Minimum Constraint Matrix Coefficient</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>4</td>
<td>Average Constraint Matrix Coefficient</td>
<td>1.875</td>
<td>1.87500</td>
</tr>
<tr>
<td>5</td>
<td>Number of Objective Nonzeros</td>
<td>3</td>
<td>3.00000</td>
</tr>
<tr>
<td>6</td>
<td>Maximum Objective Coefficient</td>
<td>4</td>
<td>4.00000</td>
</tr>
<tr>
<td>7</td>
<td>Minimum Objective Coefficient</td>
<td>2</td>
<td>2.00000</td>
</tr>
<tr>
<td>8</td>
<td>Average Objective Coefficient</td>
<td>3</td>
<td>3.00000</td>
</tr>
<tr>
<td>9</td>
<td>Number of RHS Nonzeros</td>
<td>3</td>
<td>3.00000</td>
</tr>
<tr>
<td>10</td>
<td>Maximum RHS</td>
<td>7</td>
<td>7.00000</td>
</tr>
<tr>
<td>11</td>
<td>Minimum RHS</td>
<td>4</td>
<td>4.00000</td>
</tr>
<tr>
<td>12</td>
<td>Average RHS</td>
<td>5.3333333333</td>
<td>5.333333</td>
</tr>
<tr>
<td>13</td>
<td>Maximum Number of Nonzeros per Column</td>
<td>3</td>
<td>3.00000</td>
</tr>
<tr>
<td>14</td>
<td>Minimum Number of Nonzeros per Column</td>
<td>2</td>
<td>2.00000</td>
</tr>
<tr>
<td>15</td>
<td>Average Number of Nonzeros per Column</td>
<td>2.67</td>
<td>2.666667</td>
</tr>
<tr>
<td>16</td>
<td>Maximum Number of Nonzeros per Row</td>
<td>3</td>
<td>3.00000</td>
</tr>
<tr>
<td>17</td>
<td>Minimum Number of Nonzeros per Row</td>
<td>2</td>
<td>2.00000</td>
</tr>
<tr>
<td>18</td>
<td>Average Number of Nonzeros per Row</td>
<td>2.67</td>
<td>2.666667</td>
</tr>
</tbody>
</table>

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

---

**Macro Variable _OROPTMODEL_**

The OPTMODEL procedure defines a macro variable named _OROPTMODEL_. This variable contains a character string that indicates the status of the solver upon termination. The contents of the macro variable depend on which solver was invoked. For the MILP solver, the various terms of _OROPTMODEL_ are interpreted as follows.

**STATUS**

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

- **OK** The solver terminated normally.
- **SYNTAX_ERROR** Syntax was used incorrectly.
- **DATA_ERROR** The input data was inconsistent.
- **OUT_OF_MEMORY** Insufficient memory was allocated to the solver.
Chapter 9: The Mixed Integer Linear Programming Solver

IO_ERROR A problem occurred in reading or writing data.

SEMANTIC_ERROR An evaluation error, such as an invalid operand type, was found.

ERROR The status cannot be classified into any of the preceding categories.

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

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

DECOMP The decomposition algorithm produced the solution data.

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

OPTIMAL The solution is optimal.

OPTIMAL_AGAP The solution is optimal within the absolute gap specified by the ABSOBJGAP= option.

OPTIMAL_RGAP The solution is optimal within the relative gap specified by the RELOBJGAP= option.

OPTIMAL_COND The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances due to scaling or choice of small INTTOL= value.

TARGET The solution is not worse than the target specified by the TARGET= option.

INFEASIBLE The problem is infeasible.

UNBOUNDED The problem is unbounded.

INFEASIBLE_OR_UNBOUNDED The problem is infeasible or unbounded.

BAD_PROBLEM_TYPE The problem type is unsupported by solver.

SOLUTION_LIM The solver reached the maximum number of solutions specified by the MAXSOLS= option.

NODE_LIM_SOL The solver reached the maximum number of nodes specified by the MAXNODES= option and found a solution.

NODE_LIM_NOSOL The solver reached the maximum number of nodes specified by the MAXNODES= option and did not find a solution.

TIME_LIM_SOL The solver reached the execution time limit specified by the MAXTIME= option and found a solution.

TIME_LIM_NOSOL The solver reached the execution time limit specified by the MAXTIME= option and did not find a solution.

ABORT_SOL The solver was stopped by user but still found a solution.

ABORT_NOSOL The solver was stopped by user and did not find a solution.

OUTMEM_SOL The solver ran out of memory but still found a solution.
OUTMEM_NOSOL  The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.

FAIL_SOL     The solver stopped due to errors but still found a solution.

FAIL_NOSOL   The solver stopped due to errors and did not find a solution.

OBJECTIVE   indicates the objective value obtained by the solver at termination.

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

\[ \frac{|\text{BestInteger} - \text{BestBound}|}{1 + |\text{BestBound}|} \]

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

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

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

INTEGER_INFEASIBILITY indicates the maximum (absolute) violation of the integrality of integer variables returned by the MILP solver.

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

NODES   indicates the number of nodes enumerated by the MILP solver by using the branch-and-bound algorithm.

SOLUTIONS_FOUND indicates the number of solutions that are found by the MILP solver.

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

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

SOLUTION_TIME indicates the time (in seconds) taken to solve the problem, including preprocessing time.
NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

Examples: MILP Solver

This section contains examples that illustrate the options and syntax of the MILP solver in PROC OPTMODEL. Example 9.1 illustrates the use of PROC OPTMODEL to solve an employee scheduling problem. Example 9.2 discusses a multicommodity transshipment problem with fixed charges. Example 9.3 demonstrates how to warm start the MILP solver. Example 9.4 shows the solution of an instance of the traveling salesman problem in PROC OPTMODEL. Other examples of mixed integer linear programs, along with example SAS code, are given in Chapter 14.

Example 9.1: Scheduling

The following example has been adapted from the example “A Scheduling Problem” in Chapter 4, “The LP Procedure” (SAS/OR User’s Guide: Mathematical Programming Legacy Procedures).

Scheduling is a common application area in which mixed integer linear programming techniques are used. In this example, you have eight one-hour time slots in each of five days. You have to assign four employees to these time slots so that each slot is covered every day. You allow the employees to specify preference data for each slot on each day. In addition, the following constraints must be satisfied:

- Each employee has some time slots for which he or she is unavailable (OneEmpPerSlot).
- Each employee must have either time slot 4 or time slot 5 off for lunch (EmpMustHaveLunch).
- Each employee can work at most two time slots in a row (AtMost2ConSlots).
- Each employee can work only a specified number of hours in the week (WeeklyHoursLimit).
To formulate this problem, let \( i \) denote a person, \( j \) denote a time slot, and \( k \) denote a day. Then, let \( x_{ijk} = 1 \) if person \( i \) is assigned to time slot \( j \) on day \( k \), and 0 otherwise. Let \( p_{ijk} \) denote the preference of person \( i \) for slot \( j \) on day \( k \). Let \( h_i \) denote the number of hours in a week that person \( i \) will work. The formulation of this problem follows:

\[
\begin{align*}
\text{max} & \quad \sum_{ijk} p_{ijk}x_{ijk} \\
\text{s.t.} & \quad \sum_{i} x_{ijk} = 1 \quad \forall j, k & \text{(OneEmpPerSlot)} \\
& \quad x_{i4k} + x_{i5k} \leq 1 \quad \forall i, k & \text{(EmpMustHaveLunch)} \\
& \quad x_{i,l,k} + x_{i,l+1,k} + x_{i,l+2,k} \leq 2 \quad \forall i, k, \text{ and } l \leq 6 & \text{(AtMost2ConSlots)} \\
& \quad \sum_{jk} x_{ijk} \leq h_i \quad \forall i & \text{(WeeklyHoursLimit)} \\
& \quad x_{ijk} \in \{0, 1\} \\
& \quad x_{ijk} = 0 \quad \forall i, j, k \text{ s.t. } p_{ijk} > 0 & \forall i, j, k
\end{align*}
\]

The following data set preferences gives the preferences for each individual, time slot, and day. A 10 represents the most desirable time slot, and a 1 represents the least desirable time slot. In addition, a 0 indicates that the time slot is not available. The data set maxhours gives the maximum number of hours each employee can work per week.

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

Using PROC OPTMODEL, you can model and solve the scheduling problem as follows:

```plaintext
proc optmodel;
/* read in the preferences and max hours from the data sets */
set <string,num> DailyEmployeeSlots;
set <string> Employees;
set <num> TimeSlots = (setof {<name,slot> in DailyEmployeeSlots} slot);
set <string> WeekDays = {"mon","tue","wed","thu","fri");
num WeeklyMaxHours{Employees};
num PreferenceWeights{DailyEmployeeSlots,Weekdays};
um NSlots = card(TimeSlots);
read data preferences into DailyEmployeeSlots=[name slot] 
    {day in Weekdays} <PreferenceWeights[name,slot,day] = col(day)>;
read data maxhours into Employees=[name] WeeklyMaxHours=hour;
/* declare the binary assignment variable x[i,j,k] */
var Assign{<name,slot> in DailyEmployeeSlots, day in Weekdays} binary;
/* for each p[i,j,k] = 0, fix x[i,j,k] = 0 */
for {<name,slot> in DailyEmployeeSlots, day in Weekdays:
    PreferenceWeights[name,slot,day] = 0}
    fix Assign[name,slot,day] = 0;
/* declare the objective function */
max TotalPreferenceWeight = 
    sum{<name,slot> in DailyEmployeeSlots, day in Weekdays}
        PreferenceWeights[name,slot,day] * Assign[name,slot,day];
/* declare the constraints */
con OneEmpPerSlot{slot in TimeSlots, day in Weekdays}:
    sum{name in Employees} Assign[name,slot,day] = 1;
con EmpMustHaveLunch{name in Employees, day in Weekdays}:
    Assign[name,4,day] + Assign[name,5,day] <= 1;
```

```
Example 9.1: Scheduling

```plaintext
con AtMost2ConsSlots(name in Employees, start in 1..NSlots-2, day in Weekdays):
    Assign[name,start,day] + Assign[name,start+1,day]
    + Assign[name,start+2,day] <= 2;

con WeeklyHoursLimit{name in Employees}:
    sum(slot in TimeSlots, day in Weekdays) Assign[name,slot,day]
    <= WeeklyMaxHours[name];

/* solve the model */
solve with milp;

/* clean up the solution */
for {<name,slot> in DailyEmployeeSlots, day in Weekdays}
    Assign[name,slot,day] = round(Assign[name,slot,day],1e-6);

str assigned_employee (TimeSlots, Weekdays) init '';
for {slot in TimeSlots, day in Weekdays} do;
    for {name in Employees: Assign[name,slot,day] > 0} do;
        assigned_employee[slot,day] = name;
    leave;
end;
end;

create data report from [slot]=TimeSlots
    {day in Weekdays} <col(day)=assigned_employee[slot,day]>
quit;
```

The following statements demonstrate how to use the PRINT procedure to display a schedule that shows how the eight time slots are covered for the week:

```plaintext
title 'Reported Solution';
proc print data=report;
id slot;
run;
```

The output from the preceding code is displayed in Output 9.1.1.

**Output 9.1.1**  Scheduling Reported Solution

<table>
<thead>
<tr>
<th></th>
<th>mon</th>
<th>tue</th>
<th>wed</th>
<th>thu</th>
<th>fri</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>marc</td>
<td>marc</td>
<td>marc</td>
<td>marc</td>
<td>marc</td>
</tr>
<tr>
<td>2</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
</tr>
<tr>
<td>3</td>
<td>mike</td>
<td>mike</td>
<td>bill</td>
<td>bill</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>bob</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
</tr>
<tr>
<td>5</td>
<td>marc</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
</tr>
<tr>
<td>6</td>
<td>marc</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
</tr>
<tr>
<td>7</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
<td>mike</td>
</tr>
<tr>
<td>8</td>
<td>marc</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
</tr>
</tbody>
</table>

*Reported Solution*
Example 9.2: Multicommodity Transshipment Problem with Fixed Charges

The following example has been adapted from the example “A Multicommodity Transshipment Problem with Fixed Charges” in Chapter 4, “The LP Procedure” (SAS/OR User’s Guide: Mathematical Programming Legacy Procedures).

This example illustrates the use of PROC OPTMODEL to generate a mixed integer linear program to solve a multicommodity network flow model with fixed charges. Consider a network with nodes $N$, arcs $A$, and a set $C$ of commodities to be shipped between the nodes. The commodities are defined in the data set COMMODITY_DATA, as follows:

```plaintext
title 'Multicommodity Transshipment Problem with Fixed Charges';

data commodity_data;
  do c = 1 to 4;
    output;
  end;
run;
```

Shipping cost $s_{ijc}$ is for each of the four commodities $c$ across each of the arcs $(i, j)$. In addition, there is a fixed charge $f_{ij}$ for the use of each arc $(i, j)$. The shipping costs and fixed charges are defined in the data set ARC_DATA, as follows:

```plaintext
data arc_data;
  input from $ to $ c1 c2 c3 c4 fx;
  datalines;
  farm-a Chicago 20 15 17 22 100
  farm-b Chicago 15 15 15 30 75
  farm-c Chicago 30 30 10 10 100
  farm-a StLouis 30 25 27 22 150
  farm-c StLouis 10 9 11 10 75
  Chicago NY 75 75 75 75 200
  StLouis NY 80 80 80 80 200
;run;
```

The supply (positive numbers) or demand (negative numbers) $d_{ic}$ at each of the nodes for each commodity $c$ is shown in the data set SUPPLY_DATA, as follows:

```plaintext
data supply_data;
  input node $ sd1 sd2 sd3 sd4;
  datalines;
  farm-a 100 100 40 .
  farm-b 100 200 50 50
  farm-c 40 100 75 100
  NY -150 -200 -50 -75
;run;
```
Let $x_{ijc}$ define the flow of commodity $c$ across arc $(i, j)$. Let $y_{ij} = 1$ if arc $(i, j)$ is used, and 0 otherwise. Since the total flow on an arc $(i, j)$ must be at most the total demand across all nodes $k \in \mathcal{N}$, you can define the trivial upper bound $u_{ijc}$ as

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

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

$$\min \sum_{(i,j) \in A} \sum_{c \in \mathcal{C}} s_{ijc} x_{ijc} + \sum_{(i,j) \in A} f_{ij} y_{ij}$$

s.t. $\sum_{j \in \mathcal{N} | (i,j) \in A} x_{ijc} - \sum_{j \in \mathcal{N} | (j,i) \in A} x_{jic} \leq d_{ic} \quad \forall i \in \mathcal{N}, c \in \mathcal{C}$ (balance_con)

$x_{ijc} \leq u_{ijc} y_{ij} \quad \forall (i,j) \in A, c \in \mathcal{C}$ (fixed_charge_con)

$x_{ijc} \geq 0 \quad \forall (i,j) \in A, c \in \mathcal{C}$

$y_{ij} \in \{0, 1\} \quad \forall (i,j) \in A$

Constraint (balance_con) ensures conservation of flow for both supply and demand. Constraint (fixed_charge_con) models the fixed charge cost by forcing $y_{ij} = 1$ if $x_{ijc} > 0$ for some commodity $c \in \mathcal{C}$.

The PROC OPTMODEL statements follow:
proc optmodel;
    set COMMODITIES;
    read data commodity_data into COMMODITIES={c};

    set <str,str> ARCS;
    num unit_cost {ARCS, COMMODITIES};
    num fixed_charge {ARCS};
    read data arc_data into ARCS={from to} {c in COMMODITIES}
        <unit_cost[from,to,c]=col('c'||c)> fixed_charge=fx;
    print unit_cost fixed_charge;

    set <str> NODES = union {<i,j> in ARCS} {i,j};
    num supply {NODES, COMMODITIES} init 0;
    read data supply_data nomiss into {node} {c in COMMODITIES}
        <supply[node,c]=col('sd'||c)>;
    print supply;

    var AmountShipped {ARCS, c in COMMODITIES} >= 0 <= sum {i in NODES}
        max(supply[i,c],0);

    /* UseArc[i,j] = 1 if arc (i,j) is used, 0 otherwise */
    var UseArc {ARCS} binary;

    /* TotalCost = variable costs + fixed charges */
    min TotalCost = sum {<i,j> in ARCS, c in COMMODITIES}
        unit_cost[i,j,c] * AmountShipped[i,j,c]
        + sum {<i,j> in ARCS} fixed_charge[i,j] * UseArc[i,j];

    con flow_balance {i in NODES, c in COMMODITIES}:
        sum {<(i),j> in ARCS} AmountShipped[i,j,c] -
        sum {<j,(i)> in ARCS} AmountShipped[j,i,c] <= supply[i,c];

    /* if AmountShipped[i,j,c] > 0 then UseArc[i,j] = 1 */
    con fixed_charge_def {<i,j> in ARCS, c in COMMODITIES}:
        AmountShipped[i,j,c] <= AmountShipped[i,j,c].ub * UseArc[i,j];

    solve;

    print AmountShipped;

    create data solution from [from to commodity]={<i,j> in ARCS,
        c in COMMODITIES: AmountShipped[i,j,c].sol ne 0} amount=AmountShipped;
quit;

Although the PROC LP example used \( M = 1.0e6 \) in the FIXED_CHARGE_DEF constraint that links the continuous variable to the binary variable, it is numerically preferable to use a smaller, data-dependent value. Here, the upper bound on \( AmountShipped[i,j,c] \) is used instead. This upper bound is calculated in the first VAR statement as the sum of all positive supplies for commodity \( c \). The logical condition \( AmountShipped[i,j,k].sol ne 0 \) in the CREATE DATA statement ensures that only the nonzero parts of the solution appear in the SOLUTION data set.

The problem summary, solution summary, and the output from the two PRINT statements are shown in Output 9.2.1.
### Example 9.2: Multicommodity Transshipment Problem with Fixed Charges

#### Output 9.2.1 Multicommodity Transshipment Problem with Fixed Charges Solution Summary

**Multicommodity Transshipment Problem with Fixed Charges**

The `OPTMODEL` Procedure

<table>
<thead>
<tr>
<th>[1]</th>
<th>[2]</th>
<th>[3] unit_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago NY</td>
<td>1</td>
<td>75</td>
</tr>
<tr>
<td>Chicago NY</td>
<td>2</td>
<td>75</td>
</tr>
<tr>
<td>Chicago NY</td>
<td>3</td>
<td>75</td>
</tr>
<tr>
<td>Chicago NY</td>
<td>4</td>
<td>75</td>
</tr>
<tr>
<td>StLouis NY</td>
<td>1</td>
<td>80</td>
</tr>
<tr>
<td>StLouis NY</td>
<td>2</td>
<td>80</td>
</tr>
<tr>
<td>StLouis NY</td>
<td>3</td>
<td>80</td>
</tr>
<tr>
<td>StLouis NY</td>
<td>4</td>
<td>80</td>
</tr>
<tr>
<td>farm-a Chicago 1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>farm-a Chicago 2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>farm-a Chicago 3</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>farm-a Chicago 4</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>farm-a StLouis 1</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>farm-a StLouis 2</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>farm-a StLouis 3</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>farm-a StLouis 4</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>farm-b Chicago 1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>farm-b Chicago 2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>farm-b Chicago 3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>farm-b Chicago 4</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>farm-c Chicago 1</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>farm-c Chicago 2</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>farm-c Chicago 3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>farm-c Chicago 4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>farm-c StLouis 1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>farm-c StLouis 2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>farm-c StLouis 3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>farm-c StLouis 4</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[1]</th>
<th>[2] fixed_charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago NY</td>
<td>200</td>
</tr>
<tr>
<td>StLouis NY</td>
<td>200</td>
</tr>
<tr>
<td>farm-a Chicago</td>
<td>100</td>
</tr>
<tr>
<td>farm-a StLouis</td>
<td>150</td>
</tr>
<tr>
<td>farm-b Chicago</td>
<td>75</td>
</tr>
<tr>
<td>farm-c Chicago</td>
<td>100</td>
</tr>
<tr>
<td>farm-c StLouis</td>
<td>75</td>
</tr>
</tbody>
</table>
### Output 9.2.1 continued

<table>
<thead>
<tr>
<th>supply</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NY</td>
<td>-150</td>
<td>-200</td>
<td>-50</td>
<td>-75</td>
</tr>
<tr>
<td>StLouis</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>farm-a</td>
<td>100</td>
<td>100</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>farm-b</td>
<td>100</td>
<td>200</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>farm-c</td>
<td>40</td>
<td>100</td>
<td>75</td>
<td>100</td>
</tr>
</tbody>
</table>

### Problem Summary

<table>
<thead>
<tr>
<th>Objective Sense</th>
<th>Minimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Function</td>
<td>TotalCost</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Linear</td>
</tr>
<tr>
<td>Number of Variables</td>
<td>35</td>
</tr>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>35</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
<tr>
<td>Binary</td>
<td>7</td>
</tr>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>Number of Constraints</td>
<td>52</td>
</tr>
<tr>
<td>Linear LE ((\leq))</td>
<td>52</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear GE ((\geq))</td>
<td>0</td>
</tr>
<tr>
<td>Linear Range</td>
<td>0</td>
</tr>
</tbody>
</table>

| Constraint Coefficients  | 112          |
### Output 9.2.1 continued

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solver</strong></td>
</tr>
<tr>
<td><strong>Algorithm</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Solution Status</strong></td>
</tr>
<tr>
<td><strong>Objective Value</strong></td>
</tr>
<tr>
<td><strong>Relative Gap</strong></td>
</tr>
<tr>
<td><strong>Absolute Gap</strong></td>
</tr>
<tr>
<td><strong>Primal Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Bound Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Integer Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Best Bound</strong></td>
</tr>
<tr>
<td><strong>Nodes</strong></td>
</tr>
<tr>
<td><strong>Solutions Found</strong></td>
</tr>
<tr>
<td><strong>Iterations</strong></td>
</tr>
<tr>
<td><strong>Presolve Time</strong></td>
</tr>
<tr>
<td><strong>Solution Time</strong></td>
</tr>
</tbody>
</table>
Output 9.2.1 continued

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>NY</td>
<td>1 110</td>
</tr>
<tr>
<td>Chicago</td>
<td>NY</td>
<td>2 100</td>
</tr>
<tr>
<td>Chicago</td>
<td>NY</td>
<td>3 50</td>
</tr>
<tr>
<td>Chicago</td>
<td>NY</td>
<td>4 75</td>
</tr>
<tr>
<td>StLouis</td>
<td>NY</td>
<td>1 40</td>
</tr>
<tr>
<td>StLouis</td>
<td>NY</td>
<td>2 100</td>
</tr>
<tr>
<td>StLouis</td>
<td>NY</td>
<td>3 0</td>
</tr>
<tr>
<td>StLouis</td>
<td>NY</td>
<td>4 0</td>
</tr>
<tr>
<td>farm-a</td>
<td>Chicago</td>
<td>1 10</td>
</tr>
<tr>
<td>farm-a</td>
<td>Chicago</td>
<td>2 0</td>
</tr>
<tr>
<td>farm-a</td>
<td>Chicago</td>
<td>3 0</td>
</tr>
<tr>
<td>farm-a</td>
<td>Chicago</td>
<td>4 0</td>
</tr>
<tr>
<td>farm-a</td>
<td>StLouis</td>
<td>1 0</td>
</tr>
<tr>
<td>farm-a</td>
<td>StLouis</td>
<td>2 0</td>
</tr>
<tr>
<td>farm-a</td>
<td>StLouis</td>
<td>3 0</td>
</tr>
<tr>
<td>farm-a</td>
<td>StLouis</td>
<td>4 0</td>
</tr>
<tr>
<td>farm-b</td>
<td>Chicago</td>
<td>1 100</td>
</tr>
<tr>
<td>farm-b</td>
<td>Chicago</td>
<td>2 100</td>
</tr>
<tr>
<td>farm-b</td>
<td>Chicago</td>
<td>3 0</td>
</tr>
<tr>
<td>farm-b</td>
<td>Chicago</td>
<td>4 0</td>
</tr>
<tr>
<td>farm-c</td>
<td>Chicago</td>
<td>1 0</td>
</tr>
<tr>
<td>farm-c</td>
<td>Chicago</td>
<td>2 0</td>
</tr>
<tr>
<td>farm-c</td>
<td>Chicago</td>
<td>3 50</td>
</tr>
<tr>
<td>farm-c</td>
<td>Chicago</td>
<td>4 75</td>
</tr>
<tr>
<td>farm-c</td>
<td>StLouis</td>
<td>1 40</td>
</tr>
<tr>
<td>farm-c</td>
<td>StLouis</td>
<td>2 100</td>
</tr>
<tr>
<td>farm-c</td>
<td>StLouis</td>
<td>3 0</td>
</tr>
<tr>
<td>farm-c</td>
<td>StLouis</td>
<td>4 0</td>
</tr>
</tbody>
</table>

Example 9.3: Facility Location

Consider the classic facility location problem. Given a set \( L \) of customer locations and a set \( F \) of candidate facility sites, you must decide on which sites to build facilities and assign coverage of customer demand to these sites so as to minimize cost. All customer demand \( d_i \) must be satisfied, and each facility has a demand capacity limit \( C \). The total cost is the sum of the distances \( c_{ij} \) between facility \( j \) and its assigned customer \( i \), plus a fixed charge \( f_j \) for building a facility at site \( j \). Let \( y_j = 1 \) represent choosing site \( j \) to build a facility, and 0 otherwise. Also, let \( x_{ij} = 1 \) represent the assignment of customer \( i \) to facility \( j \), and 0 otherwise. This model can be formulated as the following integer linear program:
Example 9.3: Facility Location

\[
\begin{align*}
\min \quad & \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_{j} y_{j} \\
\text{s.t.} \quad & \sum_{j \in F} x_{ij} = 1 \quad \forall i \in L \quad \text{(assign\_def)} \\
& x_{ij} \leq y_{j} \quad \forall i \in L, j \in F \quad \text{(link)} \\
& \sum_{i \in L} d_{i} x_{ij} \leq C y_{j} \quad \forall j \in F \quad \text{(capacity)} \\
& x_{ij} \in \{0, 1\} \quad \forall i \in L, j \in F \\
& y_{j} \in \{0, 1\} \quad \forall j \in F
\end{align*}
\]

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:

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

/* generate random site locations and fixed charge */
data sdata(drop=i);
   call streaminit(&seed);
   length name $8;
   do i = 1 to &NumSites;
```

The following PROC OPTMODEL statements first generate and solve the model that contains the no-fixed-charge variant of the cost function. Next, they solve the fixed-charge model and demonstrate the usage of the MAXPOOLSOLS= option for outputting multiple solutions. Note that the solution to the model that has no fixed charge is feasible for the fixed-charge model and should provide a good starting point for the MILP solver. Use the PRIMALIN option to provide an incumbent solution (warm start).

```sas
proc optmodel;
    set <str> CUSTOMERS;
    set <str> SITES init {};
    /* x and y coordinates of CUSTOMERS and SITES */
    num x {CUSTOMERS union SITES};
    num y {CUSTOMERS union SITES};
    num demand {CUSTOMERS};
    num fixed_charge {SITES};
    /* distance from customer i to site j */
    num dist {i in CUSTOMERS, j in SITES} = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);
    read data cdata into CUSTOMERS=[name] x y demand;
    read data sdata into SITES=[name] x y fixed_charge;
    var Assign {CUSTOMERS, SITES} binary;
    var Build {SITES} binary;
    min CostNoFixedCharge
        = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j];
    min CostFixedCharge
        = CostNoFixedCharge + sum {j in SITES} fixed_charge[j] * Build[j];
    /* each customer assigned to exactly one site */
    con assign_def {i in CUSTOMERS}:
        sum {j in SITES} Assign[i,j] = 1;
    /* if customer i assigned to site j, then facility must be built at j */
    con link {i in CUSTOMERS, j in SITES}:
        Assign[i,j] <= Build[j];
    /* each site can handle at most &SiteCapacity demand */
    con capacity {j in SITES}:
        sum {i in CUSTOMERS} demand[i] * Assign[i,j] <= &SiteCapacity * Build[j];
    /* solve the MILP with no fixed charges */
    solve obj CostNoFixedCharge with milp;
    /* clean up the solution */
    for {i in CUSTOMERS, j in SITES} Assign[i,j] = round(Assign[i,j]);
    for {j in SITES} Build[j] = round(Build[j]);
    call symput('varcostNo',put(CostNoFixedCharge,6.1));
    /* create a data set for use by PROC SGPLOT */
    create data CostNoFixedCharge_Data from
        [customer site]={i in CUSTOMERS, j in SITES: Assign[i,j] = 1}
        x1=x[i] y1=y[i] x2=x[j] y2=y[j]
```
function='line' drawspace='datavalue' linethickness=1 linecolor='black';
submit;
data csdata;
set cdata(rename=(y=cy)) sdata(rename=(y=sy));
run;
title1 "Facility Location Problem";
title2 "TotalCost = &varcostNo (Variable = &varcostNo, Fixed = 0)";
proc sgplot data=csdata sganno=CostNoFixedCharge_Data noautolegend;
  scatter x=x y=cy / datalabel=name datalabelattrs=(size=6pt)
             markerattrs=(symbol=circlefilled color=black size=6pt);
  scatter x=x y=sy / datalabel=name datalabelattrs=(size=6pt)
             markerattrs=(symbol=diamond color=blue size=6pt);
  xaxis display=(nolabel);
  yaxis display=(nolabel);
run;
endsubmit;
/* solve the MILP with fixed charges with warm start */
solve obj CostFixedCharge with milp / primalin maxpoolsols=3;
num varcost = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j].sol;
num fixcost = sum {j in SITES} fixed_charge[j] * Build[j].sol;
for {s in 1.._NSOL_} do;
  /* clean up the solution */
  for {i in CUSTOMERS, j in SITES} Assign[i,j] = round(Assign[i,j].sol[s]);
  for {j in SITES} Build[j] = round(Build[j].sol[s]);
  call symput('varcost', put(varcost,6.1));
  call symput('fixcost', put(fixcost,5.1));
  call symput('totalcost', put(CostFixedCharge,6.1));
  /* create a data set for use by PROC SGPLOT */
create data CostFixedCharge_Data from
  [customer site]={i in CUSTOMERS, j in SITES: Assign[i,j] = 1}
  x1=x[i] y1=y[i] x2=x[j] y2=y[j]
function='line' drawspace='datavalue' linethickness=1 linecolor='black';
submit s;
  title1 "Facility Location Problem: Solution &s";
  title2 "TotalCost = &totalcost (Variable = &varcost, Fixed = &fixcost)";
  proc sgplot data=csdata sganno=CostFixedCharge_Data noautolegend;
    scatter x=x y=cy / datalabel=name datalabelattrs=(size=6pt)
               markerattrs=(symbol=circlefilled color=black size=6pt);
    scatter x=x y=sy / datalabel=name datalabelattrs=(size=6pt)
               markerattrs=(symbol=diamond color=blue size=6pt);
    xaxis display=(nolabel);
    yaxis display=(nolabel);
run;
endsubmit;
end;
quit;

Output 9.3.1 displays the information that is printed in the log for the facility location problem.
Output 9.3.1 OPTMODEL Log for Facility Location

NOTE: Problem generation will use 16 threads.
NOTE: The problem has 510 variables (0 free, 0 fixed).
NOTE: The problem has 510 binary and 0 integer variables.
NOTE: The problem has 560 linear constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The initial MILP heuristics are applied.
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 16 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1249.9063818</td>
<td>1177.1539196</td>
<td>6.18%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP presolver is applied again.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1249.9063818</td>
<td>1177.1539196</td>
<td>6.18%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1249.9063818</td>
<td>1190.5434820</td>
<td>4.99%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>1192.6273240</td>
<td>1190.5434820</td>
<td>0.18%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP solver added 12 cuts with 273 cut coefficients at the root.
NOTE: Optimal within relative gap.
NOTE: Objective = 1192.627324.
NOTE: Problem generation will use 16 threads.
NOTE: The problem has 510 variables (0 free, 0 fixed).
NOTE: The problem uses 1 implicit variables.
NOTE: The problem has 510 binary and 0 integer variables.
NOTE: The problem has 560 linear constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The initial MILP heuristics are applied.
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 16 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>24086.8916716</td>
<td>0</td>
<td>24087</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>24086.8916716</td>
<td>19197.7909681</td>
<td>25.47%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>24086.8916716</td>
<td>19206.4233772</td>
<td>25.41%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>24086.8916716</td>
<td>19210.6103687</td>
<td>25.38%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>24086.8916716</td>
<td>19217.2330320</td>
<td>25.34%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
<td>21700.2572819</td>
<td>19217.2330320</td>
<td>12.92%</td>
<td>0</td>
</tr>
</tbody>
</table>
The output of the program is shown in Output 9.3.2.
Output 9.3.2 continued

Facility Location Problem: Solution 1
TotalCost = 21549 (Variable = 1456.5, Fixed = 20092)
Example 9.3: Facility Location

Output 9.3.2 continued

Facility Location Problem: Solution 2
TotalCost = 21553 (Variable = 1460.5, Fixed = 20092)
The economic trade-off for the fixed-charge model forces you to build fewer sites and push more demand to each site.

It is possible to expedite the solution of the fixed-charge facility location problem by choosing appropriate branching priorities for the decision variables. Recall that for each site $j$, the value of the variable $y_j$ determines whether or not a facility is built on that site. Suppose you decide to branch on the variables $y_j$ before the variables $x_{ij}$. You can set a higher branching priority for $y_j$ by using the .priority suffix for the Build variables in PROC OPTMODEL, as follows:

```plaintext
for{j in SITES} Build[j].priority=10;
```

Setting higher branching priorities for certain variables is not guaranteed to speed up the MILP solver, but it can be helpful in some instances. The following program creates and solves an instance of the facility location problem, giving higher priority to the variables $y_j$. The LOGFREQ= option is used to abbreviate the node log.
Example 9.3: Facility Location

```sas
%let NumCustomers = 45;
%let NumSites = 8;
%let SiteCapacity = 35;
%let MaxDemand = 10;
%let xmax = 200;
%let ymax = 100;
%let seed = 2345;

/* generate random customer locations */
data cdata(drop=i);
  length name $8;
  do i = 1 to &NumCustomers;
    name = compress('C'||put(i,best.));
    x = rand('UNIFORM') * &xmax;
    y = rand('UNIFORM') * &ymax;
    demand = rand('UNIFORM') * &MaxDemand;
    output;
  end;
runc;
/* 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 = rand('UNIFORM') * &xmax;
    y = rand('UNIFORM') * &ymax;
    fixed_charge = (abs(&xmax/2-x) + abs(&ymax/2-y)) / 2;
    output;
  end;
runc;
proc optmodel;
  set <str> CUSTOMERS;
  set <str> SITES init {};

  /* x and y coordinates of CUSTOMERS and SITES */
  num x {CUSTOMERS union SITES};
  num y {CUSTOMERS union SITES};
  num demand {CUSTOMERS};
  num fixed_charge {SITES};

  /* distance from customer i to site j */
  num dist {i in CUSTOMERS, j in SITES}
    = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);
  read data cdata into CUSTOMERS=[name] x y demand;
  read data sdata into SITES=[name] x y fixed_charge;

  var Assign {CUSTOMERS, SITES} binary;
  var Build {SITES} binary;
```
min CostFixedCharge
    = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j]
    + sum {j in SITES} fixed_charge[j] * Build[j];

/* each customer assigned to exactly one site */
con assign_def {i in CUSTOMERS}:
    sum {j in SITES} Assign[i,j] = 1;

/* if customer i assigned to site j, then facility must be built at j */
con link {i in CUSTOMERS, j in SITES}:
    Assign[i,j] <= Build[j];

/* each site can handle at most &SiteCapacity demand */
con capacity {j in SITES}:
    sum {i in CUSTOMERS} demand[i] * Assign[i,j] <= &SiteCapacity * Build[j];

/* assign priority to Build variables (y) */
for{j in SITES} Build[j].priority=10;

/* solve the MILP with fixed charges, using branching priorities */
solve obj CostFixedCharge with milp / logfreq=1000;
quit;

The resulting output is shown in Output 9.3.3.
Output 9.3.3  PROC OPTMODEL Log for Facility Location with Branching Priorities

NOTE: There were 45 observations read from the data set WORK.CDATA.
NOTE: There were 8 observations read from the data set WORK.SDATA.
NOTE: Problem generation will use 16 threads.
NOTE: The problem has 368 variables (0 free, 0 fixed).
NOTE: The problem has 368 binary and 0 integer variables.
NOTE: The problem has 413 linear constraints (368 LE, 45 EQ, 0 GE, 0 range).
NOTE: The problem has 1448 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 368 variables, 413 constraints, and 1448 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 16 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2010.8409428</td>
<td>1856.1887172</td>
<td>8.33%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2010.8409428</td>
<td>1865.6935656</td>
<td>7.78%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1969.5351860</td>
<td>1865.6935656</td>
<td>5.57%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1969.5351860</td>
<td>1869.5997552</td>
<td>5.35%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1969.5351860</td>
<td>1872.7829403</td>
<td>5.17%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1969.5351860</td>
<td>1872.7829403</td>
<td>5.17%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1875.5427075</td>
<td>3.51%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1877.3145846</td>
<td>3.41%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1880.6572671</td>
<td>3.23%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1881.1243727</td>
<td>3.20%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1881.2709632</td>
<td>3.20%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1881.2863228</td>
<td>3.20%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1882.4166946</td>
<td>3.13%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1883.1263478</td>
<td>3.09%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1883.3726047</td>
<td>3.08%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1884.0841457</td>
<td>3.04%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1884.6919702</td>
<td>3.01%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1885.0059762</td>
<td>3.00%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1885.0544672</td>
<td>2.99%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1885.0768300</td>
<td>2.99%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1885.1502580</td>
<td>2.98%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP presolver is applied again.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1969.5351860</td>
<td>1872.7829403</td>
<td>5.17%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1969.5351860</td>
<td>1872.7829403</td>
<td>5.17%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1872.7829403</td>
<td>3.66%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1872.7829403</td>
<td>3.66%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1873.4826852</td>
<td>3.63%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1875.5427075</td>
<td>3.51%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1877.3145846</td>
<td>3.41%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1880.6572671</td>
<td>3.23%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1881.1243727</td>
<td>3.20%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1881.2709632</td>
<td>3.20%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1881.2863228</td>
<td>3.20%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1882.4166946</td>
<td>3.13%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1883.1263478</td>
<td>3.09%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1883.3726047</td>
<td>3.08%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1884.0841457</td>
<td>3.04%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1884.6919702</td>
<td>3.01%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1884.8805876</td>
<td>3.00%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1884.9934467</td>
<td>2.99%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1885.0168380</td>
<td>2.99%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1941.3999471</td>
<td>1885.1502580</td>
<td>2.98%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP solver added 23 cuts with 700 cut coefficients at the root.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>8</td>
<td>8</td>
<td>1923.1420558</td>
<td>1889.4162078</td>
<td>1.78%</td>
<td>0</td>
</tr>
<tr>
<td>162</td>
<td>61</td>
<td>9</td>
<td>1922.3068612</td>
<td>1894.8107723</td>
<td>1.45%</td>
<td>0</td>
</tr>
</tbody>
</table>
Example 9.4: Traveling Salesman Problem

The traveling salesman problem (TSP) is that of finding a minimum cost tour in an undirected graph $G$ with vertex set $V = \{1, \ldots, |V|\}$ and edge set $E$. A tour is a connected subgraph for which each vertex has degree two. The goal is then to find a tour of minimum total cost, where the total cost is the sum of the costs of the edges in the tour. With each edge $e \in E$ we associate a binary variable $x_e$, which indicates whether edge $e$ is part of the tour, and a cost $c_e \in \mathbb{R}$. Let $\delta(S) = \{i, j \in E \mid i \in S, j \notin S\}$. Then an integer linear programming (ILP) formulation of the TSP is as follows:

$$\min \sum_{e \in E} c_e x_e$$

subject to:

$$\sum_{e \in \delta(i)} x_e = 2 \quad \forall i \in V \quad \text{(two_match)}$$

$$\sum_{e \in \delta(S)} x_e \geq 2 \quad \forall S \subseteq V, 2 \leq |S| \leq |V| - 1 \quad \text{(subtour_elim)}$$

$$x_e \in \{0, 1\} \quad \forall e \in E$$

The equations (two_match) are the matching constraints, which ensure that each vertex has degree two in the subgraph, while the inequalities (subtour_elim) are known as the subtour elimination constraints (SECs) and enforce connectivity.

Since there is an exponential number $O(2^{|V|})$ of SECs, it is impossible to explicitly construct the full TSP formulation for large graphs. An alternative formulation of polynomial size was introduced by Miller, Tucker, and Zemlin (1960) (MTZ):

$$\min \sum_{(i,j) \in E} c_{ij} x_{ij}$$

subject to:

$$\sum_{j \in V} x_{ij} = 1 \quad \forall i \in V \quad \text{(assign_i)}$$

$$\sum_{i \in V} x_{ij} = 1 \quad \forall j \in V \quad \text{(assign_j)}$$

$$u_i - u_j + 1 \leq (|V| - 1)(1 - x_{ij}) \quad \forall (i, j) \in E, i \neq 1, j \neq 1 \quad \text{(mtz)}$$

$$2 \leq u_i \leq |V| \quad \forall i \in \{2, \ldots, |V|\},$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in E$$

This formulation uses a directed graph. Constraints (assign_i) and (assign_j) now enforce that each vertex has degree two (one edge in, one edge out). The MTZ constraints (mtz) enforce that no subtours exist.

TSPLIB is a set of benchmark instances for the TSP. For more information about TSPLIB, see https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/. The following DATA step converts a TSPLIB instance of type EUC_2D into a SAS data set that contains the coordinates of the vertices:
Example 9.4: Traveling Salesman Problem

/* convert the TSPLIB instance into a data set */
data tspData(drop=H);
   infile "st70.tsp";
   input H $1. @;
   if H not in ('N','T','C','D','E');
   input @1 var1-var3;
run;

The following PROC OPTMODEL statements attempt to solve the TSPLIB instance st70.tsp by using the MTZ formulation:

/* direct solution using the MTZ formulation */
proc optmodel;
   set VERTICES;
   set EDGES = {i in VERTICES, j in VERTICES: i ne j};
   num xc {VERTICES};
   num yc {VERTICES};
   /* read in the instance and customer coordinates (xc, yc) */
   read data tspData into VERTICES=[_n_] xc=var2 yc=var3;
   /* the cost is the Euclidean distance rounded to the nearest integer */
   num c {<i,j> in EDGES}
      init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);
   var x {EDGES} binary;
   var u {i in 2..card(VERTICES)} >= 2 <= card(VERTICES);
   /* each vertex has exactly one in-edge and one out-edge */
   con assign_i {i in VERTICES}:
      sum {j in VERTICES: i ne j} x[i,j] = 1;
   con assign_j {j in VERTICES}:
      sum {i in VERTICES: i ne j} x[i,j] = 1;
   /* minimize the total cost */
   min obj
      = sum {<i,j> in EDGES} (if i > j then c[i,j] else c[j,i]) * x[i,j];
   /* no subtours */
   con mtz {<i,j> in EDGES : (i ne 1) and (j ne 1)}:
      u[i] - u[j] + 1 <= (card(VERTICES) - 1) * (1 - x[i,j]);
   solve with milp / maxtime = 600;
quit;

It is well known that the MTZ formulation is much weaker than the subtour formulation. The exponential number of SECs makes it impossible, at least in large instances, to use a direct call to the MILP solver with the subtour formulation. For this reason, if you want to solve the TSP with one SOLVE statement, you must use the MTZ formulation and rely strictly on generic cuts and heuristics. Except for very small instances, this is unlikely to be a good approach.

A much more efficient way to tackle the TSP is to dynamically generate the subtour inequalities as cuts. Typically this is done by solving the LP relaxation of the two-matching problem, finding violated subtour cuts, and adding them iteratively. The problem of finding violated cuts is known as the separation problem. In this case, the separation problem takes the form of a minimum cut problem, which is nontrivial to implement efficiently. Therefore, for the sake of illustration, an integer program is solved at each step of the process.

The initial formulation of the TSP is the integral two-matching problem. You solve this by using PROC OPTMODEL to obtain an integral matching, which is not necessarily a tour. In this case, the separation problem is trivial. If the solution is a connected graph, then it is a tour, so the problem is solved. If the
solution is a disconnected graph, then each component forms a violated subtour constraint. These constraints are added to the formulation, and the integer program is solved again. This process is repeated until the solution defines a tour.

The following PROC OPTMODEL statements solve the TSP by using the subtour formulation and iteratively adding subtour constraints:

```plaintext
/* iterative solution using the subtour formulation */
proc optmodel;
  set VERTICES;
  set EDGES = {i in VERTICES, j in VERTICES: i > j};
  num xc {VERTICES};
  num yc {VERTICES};

  num numsubtour init 0;
  set SUBTOUR {1..numsubtour};

  /* read in the instance and customer coordinates (xc, yc) */
  read data tspData into VERTICES=[var1] xc=var2 yc=var3;

  /* the cost is the Euclidean distance rounded to the nearest integer */
  num c {<i,j> in EDGES}
      init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);

  var x {EDGES} binary;

  /* minimize the total cost */
  min obj =
      sum {<i,j> in EDGES} c[i,j] * x[i,j];

  /* each vertex has exactly one in-edge and one out-edge */
  con two_match {i in VERTICES}:
      sum {j in VERTICES: i > j} x[i,j]
      + sum {j in VERTICES: i < j} x[j,i] = 2;

  /* no subtours (these constraints are generated dynamically) */
  con subtour_elim {s in 1..numsubtour}:
      sum {<i,j> in EDGES: (i in SUBTOUR[s] and j not in SUBTOUR[s])
          or (i not in SUBTOUR[s] and j in SUBTOUR[s])} x[i,j] >= 2;

  /* this starts the algorithm to find violated subtours */
  set <num,num> EDGES1;
  set VERTICES1 = union{<i,j> in EDGES1} {i, j};
  num component {VERTICES1};
  num numcomp init 2;
  num iter init 1;
  num numiters init 1;
  set ITERS = 1..numiters;
  num sol {ITERS, EDGES};

  /* initial solve with just matching constraints */
  solve;
  call symput(compress('obj'||put(iter,best.)),
          trim(left(put(round(obj),best.))));
```
for \( \langle i, j \rangle \) in EDGES \[\text{sol}[\text{iter},i,j] = \text{round}(x[i,j])\];

/* while the solution is disconnected, continue */
do while (numcomp > 1);
   iter = iter + 1;

   /* find connected components of support graph */
EDGES1 = \{\langle i, j \rangle \in EDGES \mid \text{round}(x[i,j].sol) = 1\};
solve with network /
   links = (include=EDGES1)
   nodes = (include=VERTICES1)
   concomp
   out = (concomp=component);
   numcomp = _oroptmodel_num_("NUM_COMPONENTS");
if numcomp = 1 then leave;
   numiters = iter;
   numsubtour = numsubtour + numcomp;
   for \{comp in 1..\text{numcomp}\} do;
      SUBTOUR\{numsubtour-numcomp+comp\}
         = \{i \in VERTICES \mid \text{component}[i] = \text{comp}\};
   end;
   solve;
   call symput(compress('obj' || put(iter,best.)),
      trim(left(put(round(obj),best.))));
   for \{\langle i, j \rangle \in EDGES\} \[\text{sol}[\text{iter},i,j] = \text{round}(x[i,j])\];
end;

You can generate plots of the solution and objective value at each stage by using the following statements:

%macro plotTSP;
%do i = 1 %to &numiters;
   /* create annotate data set to draw subtours */
data anno(drop=iter);
   retain drawspace 'datavalue' linethickness 1 function 'line';
   set solData;
   where iter = &i;
run;

title1 h=2 "TSP: Iter = &i, Objective = &obj\&i";
title2;

   proc sgplot data=tspData sganno=anno;
      scatter x=var2 y=var3 / datalabel=var1;
xaxis display=none;
%mend;
The plot in Output 9.4.1 shows the solution and objective value at each stage. Notice that each stage restricts some subset of subtours. When you reach the final stage, you have a valid tour.

**NOTE:** See the “Examples” section in Chapter 10, “The Network Solver,” for an example of how to use a specialized algorithm to solve the TSP.

**Output 9.4.1** Traveling Salesman Problem Iterative Solution

![Output 9.4.1 Traveling Salesman Problem Iterative Solution](image-url)
References


Subject Index

active nodes
  OPTMODEL procedure, MILP solver, 375

branch-and-bound
  control options, 376
  branching priorities
    OPTMODEL procedure, MILP solver, 378
  branching variable
    OPTMODEL procedure, MILP solver, 375

cutting planes
  OPTMODEL procedure, MILP solver, 378

decomposition algorithm
  MILP solver, 373

facility location
  MILP solver examples, 396

OROPTMODEL
  _OROPTMODEL_, 383

MILP solver
  problem statistics, 382

MILP solver examples
  branching priorities, 404
  facility location, 396
  multicommodity problems, 390
  scheduling, 386
  traveling salesman problem, 408

multicommodity problems
  MILP solver examples, 390

node selection
  OPTMODEL procedure, MILP solver, 376

number of threads, 375

OPTMODEL procedure, MILP solver
  active nodes, 375
  branch-and-bound, 376
  branching priorities, 378
  branching variable, 375
  cutting planes, 378
  functional summary, 363
  introductory example, 362
  node selection, 376
  _OROPTMODEL_ macro variable, 383
  presolve, 378
  probing, 378
  variable selection, 377

presolve
  OPTMODEL procedure, MILP solver, 378

probing
  OPTMODEL procedure, MILP solver, 378

random seed, 368

scheduling
  MILP solver examples, 386

SOLVE WITH MILP
  number of threads, 375

traveling salesman problem
  MILP solver examples, 408

variable selection
  OPTMODEL procedure, MILP solver, 377
Syntax Index

ABSOBJGAP= option
  SOLVE WITH MILP statement, 366

CONCURRENT= option
  SOLVE WITH MILP statement, 374

CONFLICTSEARCH= option
  SOLVE WITH MILP statement, 369

CUTCLIQUE= option
  SOLVE WITH MILP statement, 372

CUTFLOWCOVER= option
  SOLVE WITH MILP statement, 372

CUTFLOWPATH= option
  SOLVE WITH MILP statement, 372

CUTFLOWPATH= option
  SOLVE WITH MILP statement, 372

CUTFLOWPATH= option
  SOLVE WITH MILP statement, 372

CUTGOMORY= option
  SOLVE WITH MILP statement, 372

CUTGUB= option
  SOLVE WITH MILP statement, 372

CUTIMPLIED= option
  SOLVE WITH MILP statement, 372

CUTKNAPSACK= option
  SOLVE WITH MILP statement, 373

CUTLAF= option
  SOLVE WITH MILP statement, 373

CUTMILIFTED= option
  SOLVE WITH MILP statement, 373

CUTMIR= option
  SOLVE WITH MILP statement, 373

CUTMULTICOMMODITY= option
  SOLVE WITH MILP statement, 373

CUTOFF= option
  SOLVE WITH MILP statement, 366

CUTS= option
  SOLVE WITH MILP statement, 373

CUTSFACCTOR= option
  SOLVE WITH MILP statement, 373

CUTSTRATEGY= option
  SOLVE WITH MILP statement, 373

CUTZEROHALF= option
  SOLVE WITH MILP statement, 373

DECOMPMASTERIP=() option
  SOLVE WITH MILP statement, 374

DECOMPMASTER=() option
  SOLVE WITH MILP statement, 374

DECOMP=() option
  SOLVE WITH MILP statement, 373

DECOMPUSUBPROB=() option
  SOLVE WITH MILP statement, 374

DETERMINISTIC= option

DISTRIBUTED= option
  SOLVE WITH MILP statement, 374

EMPHASIS= option
  SOLVE WITH MILP statement, 366

FEASTOL= option
  SOLVE WITH MILP statement, 366

HEURISTICS= option
  SOLVE WITH MILP statement, 369

INTTTOL= option
  SOLVE WITH MILP statement, 367

LOGFREQ= option
  SOLVE WITH MILP statement, 367

LOGLEVEL= option
  SOLVE WITH MILP statement, 367

MAXNODES= option
  SOLVE WITH MILP statement, 367

MAXPOOLSOLS= option
  SOLVE WITH MILP statement, 367

MAXSOLS= option
  SOLVE WITH MILP statement, 368

MAXTIME= option
  SOLVE WITH MILP statement, 368

NODESEL= option
  SOLVE WITH MILP statement, 370

NTHREADS= option
  SOLVE WITH MILP statement, 375

OPTMODEL procedure, MILP solver, 363

OPTTOL= option
  SOLVE WITH MILP statement, 368

PRESOLVER= option
  SOLVE WITH MILP statement, 365

PRIMALIN option
  SOLVE WITH MILP statement, 366

PRINTFREQ= option
  SOLVE WITH MILP statement, 367

PRIORITY= option
  SOLVE WITH MILP statement, 370

PROBE= option
  SOLVE WITH MILP statement, 368
RELOBJGAP= option
  SOLVE WITH MILP statement, 368
RESTARTS= option
  SOLVE WITH MILP statement, 370

SCALE= option
  SOLVE WITH MILP statement, 368
SEED= option
  SOLVE WITH MILP statement, 368

SOLVE WITH MILP statement
  ABSOBJGAP= option, 366
  CONCURRENT= option, 374
  CONFLICTSEARCH= option, 369
  CUTCLIQUE= option, 372
  CUTFLOWCOVER= option, 372
  CUTFLOWPATH= option, 372
  CUTGOMORY= option, 372
  CUTGUB= option, 372
  CUTIMPLIED= option, 372
  CUTKNAPSACK= option, 373
  CUTFLOWLIFTED= option, 373
  CUTMIR= option, 373
  CUTMULTICOMMODITY= option, 373
  CUTOFF= option, 366
  CUTS= option, 373
  CUTSFAC= option, 373
  CUTSTRATEGY= option, 373
  CUTFLOWER= option, 373
  DECOMPMASTER= option, 374
  DECOMPMASTER=() option, 374
  DECOMP=() option, 373
  DECOMPSUBPROB=() option, 374
  DETERMINISTIC= option, 374
  DISTRIBUTED= option, 374
  EMPHASIS= option, 366
  FEASTOL= option, 366
  HEURISTICS= option, 369
  INTTOL= option, 367
  LOGFREQ= option, 367
  LOGLEVEL= option, 367
  MAXNODES= option, 367
  MAXPOOL= option, 367
  MAXSOLS= option, 368
  MAXTIME= option, 368
  NODESEL= option, 370
  NTHREADS= option, 375
  OPTTOL= option, 368
  PRESOLVE= option, 365
  PRIMALIN option, 366
  PRINTFREQ= option, 367
  PRIORITY= option, 370
  PROBE= option, 368
  RELOBJGAP= option, 368
  RESTARTS= option, 370
  SCALE= option, 368
  SEED= option, 368
  STRONGITER= option, 370
  STRONGLEN= option, 371
  SYMMETRY= option, 371
  TARGET= option, 369
  TIMETYPE= option, 369
  VARSEL= option, 371
  STRONGITER= option
    SOLVE WITH MILP statement, 370
  STRONGLEN= option
    SOLVE WITH MILP statement, 371
  SYMMETRY= option
    SOLVE WITH MILP statement, 371
  TARGET= option
    SOLVE WITH MILP statement, 369
  TIMETYPE= option
    SOLVE WITH MILP statement, 369
  VARSEL= option
    SOLVE WITH MILP statement, 371