Paper SAS1502-2015

Using the OPTMODEL Procedure in SAS/OR® to Solve Complex Problems

Rob Pratt, SAS Institute Inc.

ABSTRACT

Mathematical optimization is a powerful paradigm for modeling and solving business problems that involve interrelated decisions about resource allocation, pricing, routing, scheduling, and similar issues. The OPTMODEL procedure in SAS/OR® software provides unified access to a wide range of optimization solvers and supports both standard and customized optimization algorithms. This paper illustrates PROC OPTMODEL's power and versatility in building and solving optimization models and describes the significant improvements that result from PROC OPTMODEL's many new features. Highlights include the recently added support for the network solver, the constraint programming solver, and the COFOR statement, which allows parallel execution of independent solver calls. Best practices for solving complex problems that require access to more than one solver are also demonstrated.

RELEVANT NEW FEATURES

This paper uses a difficult optimization problem, described in the next section, to illustrate several features recently added to the OPTMODEL procedure in SAS/OR. In this section, the features are described generally.

The first feature is the SUBMIT block, introduced in SAS/OR 12.1. The SUBMIT block gives you access to the full SAS® system from within a PROC OPTMODEL session, enabling you to call other procedures or DATA steps without exiting PROC OPTMODEL. The following statements show the syntax, with SUBMIT and ENDSUBMIT statements enclosing the block of arbitrary SAS code:

```
proc optmodel;
   ...
   submit [arguments] [/ options];
      [arbitrary SAS code]
   endsubmit;
   ...
quit;
```

In SAS/OR, you also have access to several network algorithms. In SAS/OR 12.1, you can use PROC OPTNET, a specialized procedure that accepts nodes and links data sets, to access these network algorithms; or you can access them from within PROC OPTMODEL by using a SUBMIT block. In SAS/OR 13.1, you have more direct access via PROC OPTMODEL's new SOLVE WITH NETWORK statement. Some of the network algorithms are diagnostic, like connected components and maximal cliques. Others solve classical network optimization problems, like minimum-cost network flow, linear assignment, and the traveling salesman problem. For more information, see the network solver chapter in *SAS/OR User's Guide: Mathematical Programming*.

For constraint programming, you can use the CLP procedure, which again is accessible from PROC OPTMODEL via a SUBMIT block. In SAS/OR 13.2, released in August 2014, you have more direct access via the new SOLVE WITH CLP statement. The CLP solver can return all or a specified number of feasible solutions. It also supports strict inequalities ($<$, $>$) and disequalities ($\neq$). Furthermore, you can express constraints that use the six predicates ALLDIFF, ELEMENT, GCC, LEXICO, PACK, and REIFY. For example, the ALLDIFF predicate forces the specified set of variables to take different values, and the REIFY predicate enables you to link a binary variable to a linear constraint. These predicates enable the CLP solver to use specialized constraint propagation techniques that exploit the problem structure. For more information, see the constraint programming solver chapter in *SAS/OR User's Guide: Mathematical Programming*.

The last new feature is the COFOR statement, which enables you to solve independent optimization problems concurrently by using multiple threads. The COFOR syntax is very similar to the serial FOR loop syntax, with just one keyword change from FOR to COFOR:

```
cofor {i in ISET} do;
   ...
   solve ...;
   ...
end;
```

A best practice is to first develop your code by using a FOR loop, and then when everything is working correctly, switch the keyword FOR to COFOR to boost the performance.

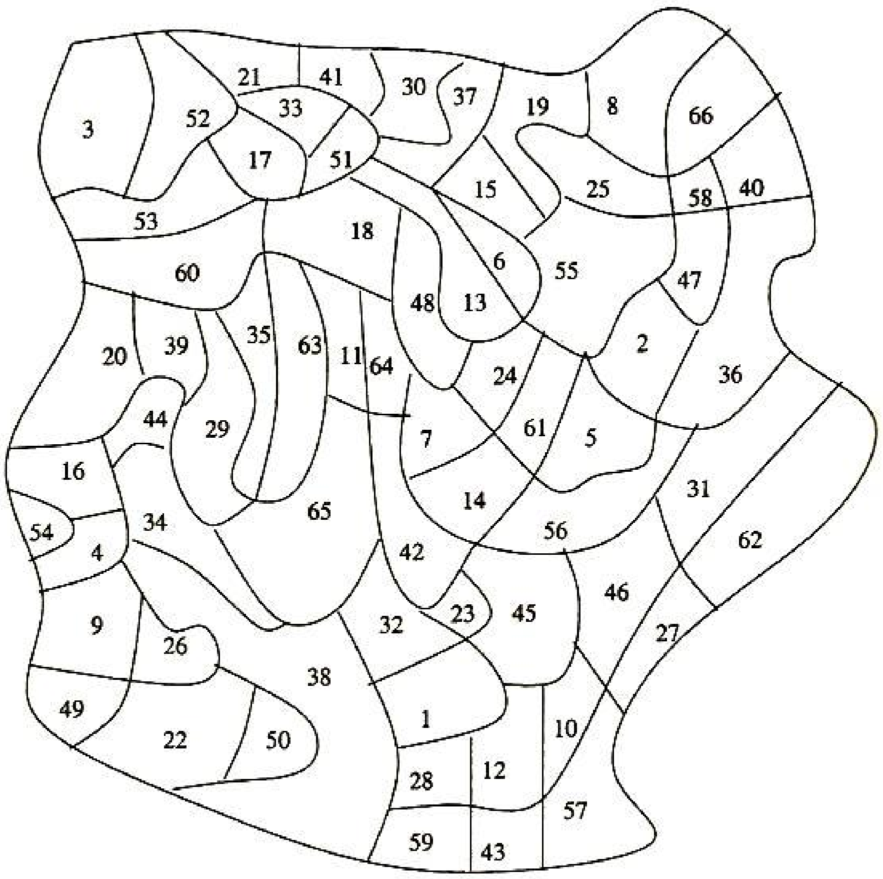
PROBLEM DESCRIPTION

The optimization problem that this paper uses to illustrate these new features is called the graph partitioning problem with connectivity constraints. You are given a graph with node weights, an integer k, and a target t within an interval from ℓ to u. The problem is to partition the nodes into k subsets (or clusters) so that the total node weight of each cluster in the partition is within the range $[\ell, u]$. In graph partitioning, the usual objective is to minimize the total weight of edges that cross clusters. But this example considers two other objectives that balance the partition by minimizing the distance from the target t. Both of these objectives are easily linearized. What makes the problem hard is the additional requirement that each cluster must be connected; that is, there must be a path between every pair of nodes in the subgraph induced by the cluster.

The author first saw this problem several years ago in a puzzle book (Shasha 2002), where it is described in terms of political districting. This connectivity requirement also arose in a recent SAS/OR consulting engagement in which service workers were required to be assigned to geographically contiguous regions. Connectivity constraints also appear in the forestry industry, where clear-cutting is subject to environmental regulations (Carvajal et al. 2013).

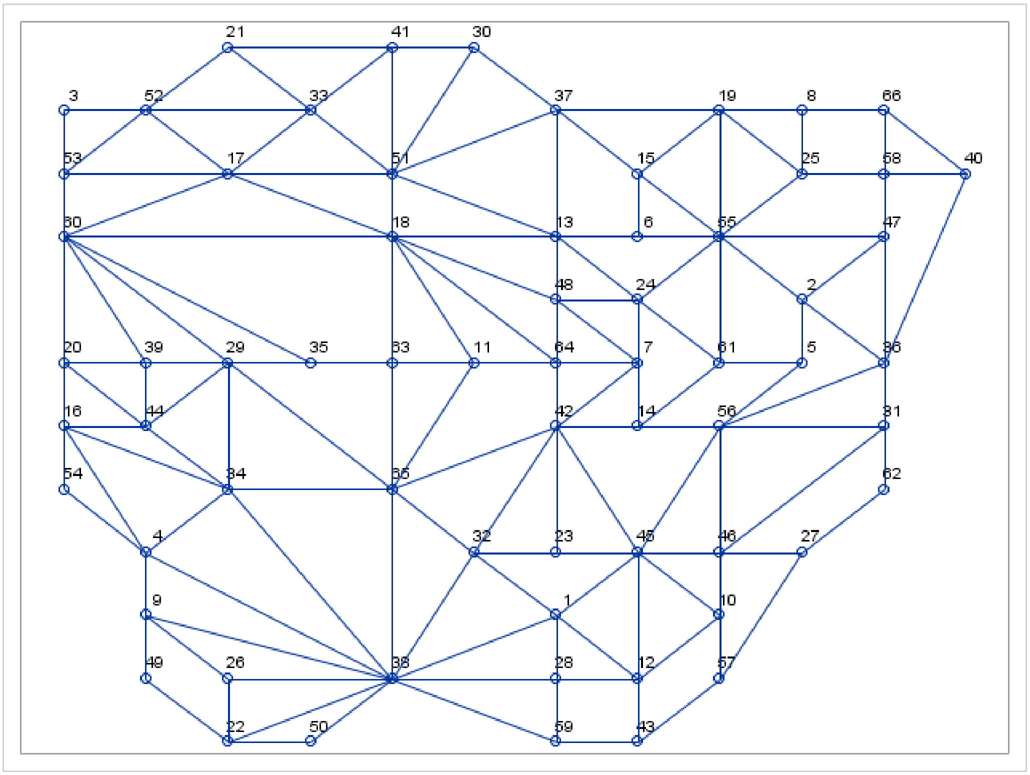
Figure 1 shows the map from Shasha (2002), which contains 66 regions.

Figure 1 Map from Shasha (2002)



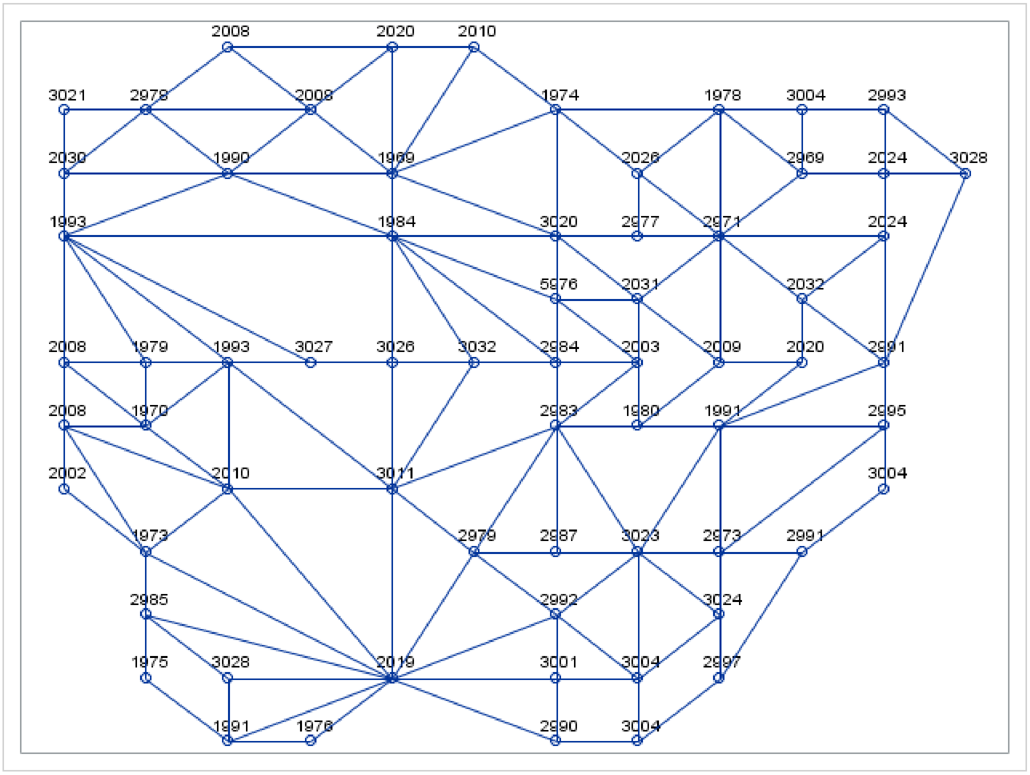
You can represent the map in terms of a graph by including a node for each region, with an edge between two nodes if the two regions share a border, as shown in Figure 2.

Figure 2 Map Represented as Graph



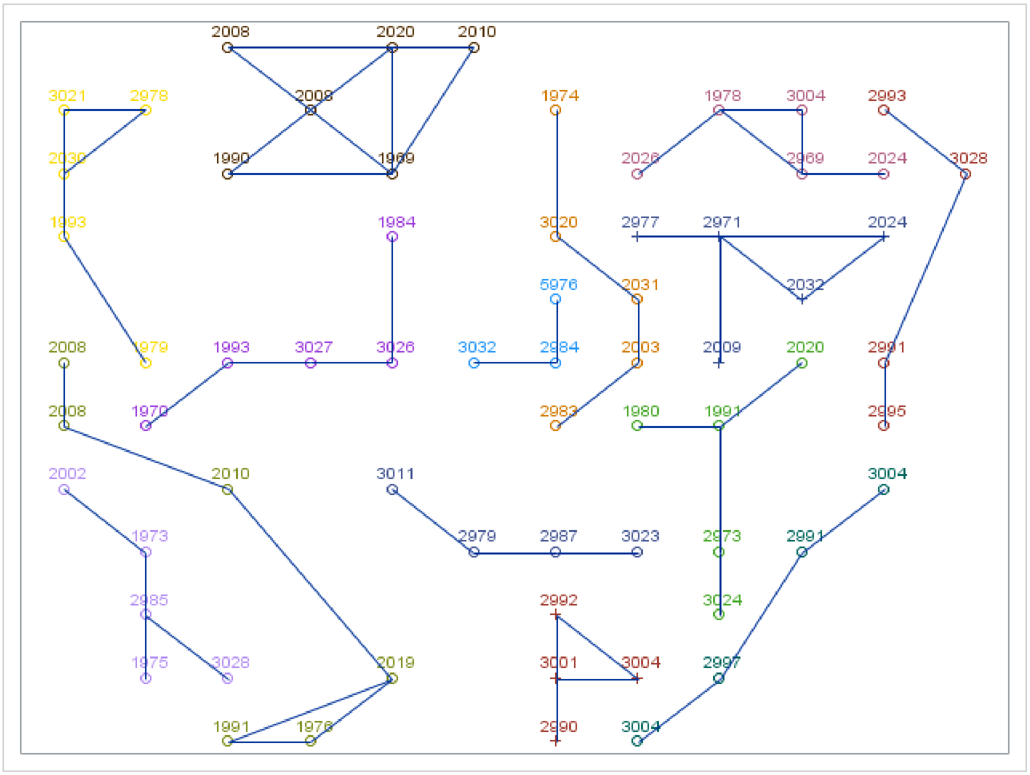
In the political districting context, the node weights are the populations of the regions, as shown in Figure 3.

Figure 3 Node Weights (Populations)



The PROC SGPLOT output in Figure 4 shows a feasible solution if the number of clusters is 14 and the population range for each cluster is 12,000 ± 100.

Figure 4 Feasible Solution for 14 Clusters, Bounds 12,000 ± 100



SOLUTION APPROACH

Several natural solution approaches for this problem use mixed integer linear programming (MILP). You can enforce connectivity in a compact MILP formulation by introducing binary variables, flow variables, and flow balance constraints, but this approach usually does not perform well. An indirect method is to apply row generation, dynamically adding constraints that exclude any disconnected subsets that appear. Another indirect approach is to apply column generation, dynamically generating connected subsets. For this example, the best method turns out to be static column generation, where all possible connected subsets are generated at once at the beginning.

STATIC COLUMN GENERATION

Suppose you have already generated all possible connected subsets that have populations in the range 12,000 ± 100. The following PROC OPTMODEL statements define the resulting cardinality-constrained set partitioning problem:

```
/* master problem: partition nodes into connected subsets,
   minimizing deviation from target */
/* UseSubset[s] = 1 if subset s is used, 0 otherwise */
var UseSubset {SUBSETS} binary;

/* L_1 norm */
num subset_population {s in SUBSETS} = sum {i in NODES_s[s]} p[i];
num subset_deviation  {s in SUBSETS} = abs(subset_population[s] - target);

min Objective1 = sum {s in SUBSETS} subset_deviation[s] * UseSubset[s];

/* each node belongs to exactly one subset */
con Partition {i in NODES}:
   sum {s in SUBSETS: i in NODES_s[s]} UseSubset[s] = 1;
/* use exactly the allowed number of subsets */
con Cardinality:
   sum {s in SUBSETS} UseSubset[s] = num_clusters;
```

The binary variable **UseSubset[s]** indicates whether subset s is used or not. One objective is to minimize the sum of absolute deviations from the target of 12,000. The Partition constraint forces each node to appear in exactly one subset. The Cardinality constraint forces the specified number of subsets to be used. Because various problems will be solved within one PROC OPTMODEL session, it is useful to name this problem. The following PROBLEM statement declares a problem named Master, in which **UseSubset** are the variables, **Objective1** is the objective, and Partition and Cardinality are the constraints:

```
   problem Master include
      UseSubset Objective1 Partition Cardinality;
```

The static column generation approach has two main steps: first generate all possible connected subsets whose weights are close enough to the target, and then solve the master problem just described. The fact that you want all possible subsets suggests the use of the CLP solver, and the fact that you want them to be connected suggests the use of the network solver.

SOLVER CALLS

The following five solver calls are used in PROC OPTMODEL to solve the problem:

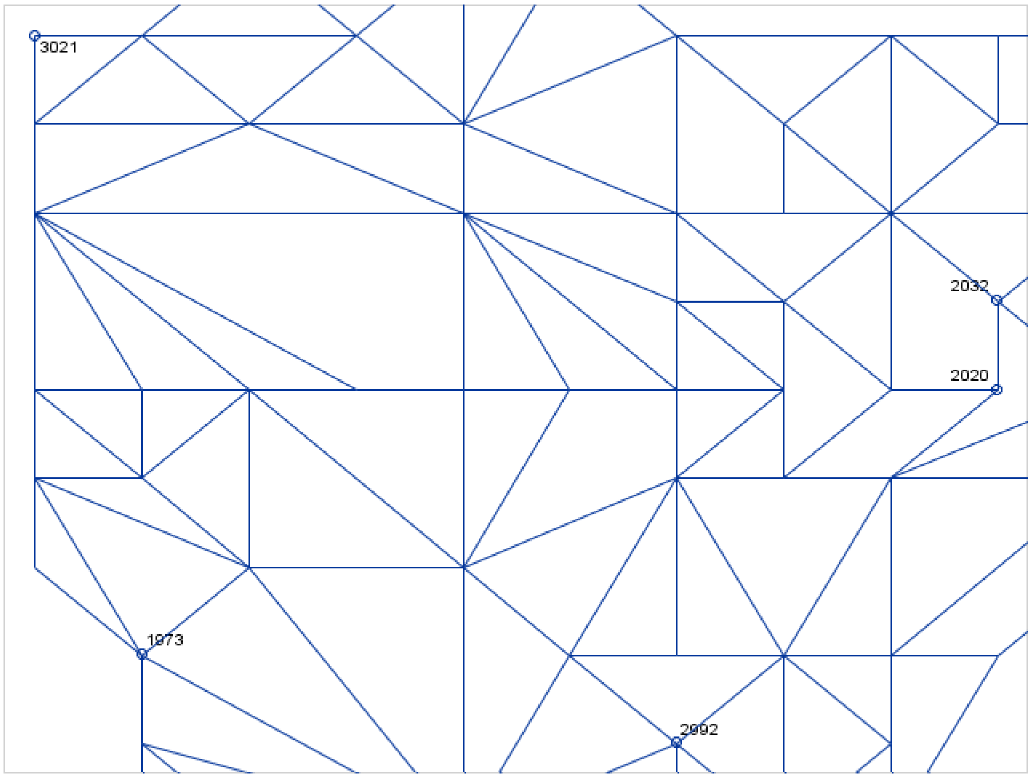
1. The MILP solver computes the maximum subset size for each node, and these values are used to generate some valid cuts that encourage connectivity. Because these problems (one for each node) are independent, you can use a COFOR loop to solve them concurrently.
2. The network solver (in particular, the all-pairs shortest path algorithm) is used to generate additional valid cuts.
3. The CLP solver generates subsets, and access to multiple solutions is demonstrated.
4. The network solver is called again (this time, invoking the connected components algorithm) to check connectivity of these subsets. Again, a COFOR loop is used to solve these independent problems concurrently.
5. The MILP solver is called to solve the master set partitioning problem declared earlier.

The subsequent sections describe these uses of the solvers in more detail and show the corresponding PROC OPTMODEL statements.

COLUMN GENERATION SUBPROBLEM

The column generation subproblem is as follows. Given the node populations p_i and a target population range $[\ell, u]$, find all connected subsets whose population is within that range. A binary decision variable x_i indicates whether or not node i appears in the subset. The population constraint is expressed as a linear range constraint $\ell \le \sum_{i \in N} p_i x_i \le u$ over these binary variables. Because connectivity is a difficult requirement, you can temporarily relax it. Completely ignoring connectivity would yield a prohibitively large number of subsets, most of which are disconnected and have to be discarded. For example, Figure 5 shows a five-node subset that satisfies the population bounds but is disconnected.

Figure 5 $\{1, 2, 3, 4, 5\}$ Satisfies Population Bounds but Is Disconnected



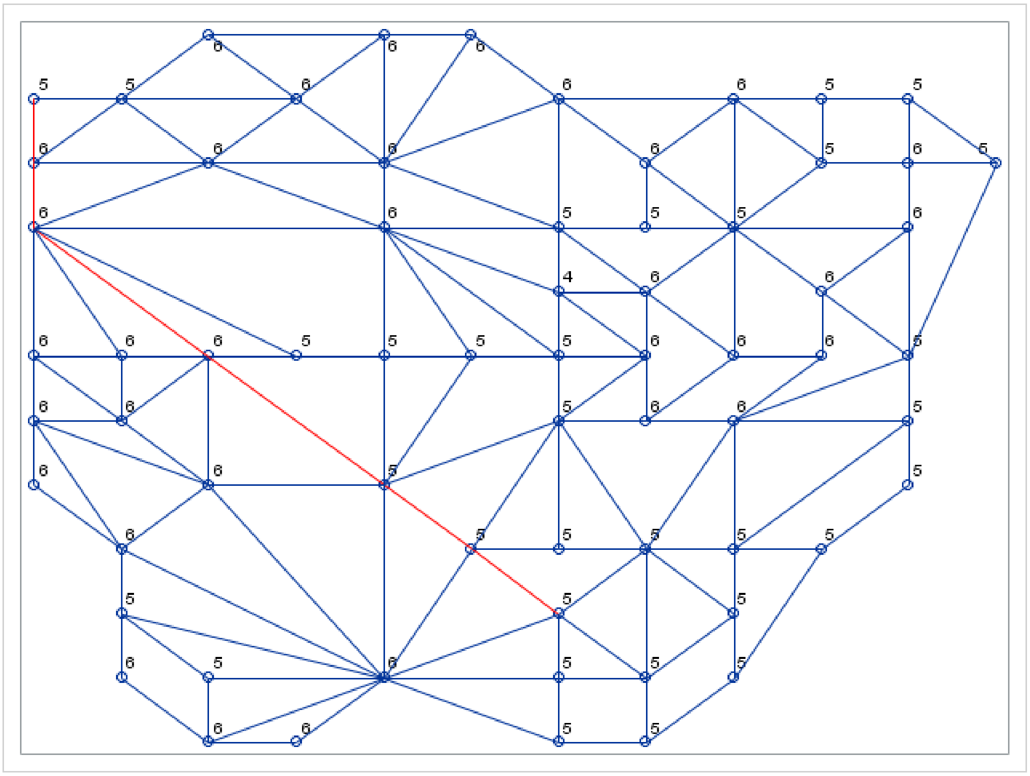
Valid Cuts

One way to encourage connectivity is to add valid cuts that disallow pairs of nodes that are too far apart from appearing in the same subset. Suppose you know that the largest number of nodes in a subset that contains node i is m_i, and suppose you also know the shortest path distances d_{ij} (number of edges) between all pairs of nodes. If $d_{ij} \geq \min(m_i, m_j)$, then nodes i and j cannot appear together in a subset, and you can enforce that rule by using the following conflict constraint:

$$x_i + x_j \leq 1 \qquad (1)$$

Figure 6 shows the values of m_i. For the endpoints i and j of the shortest path shown in red, $m_i = m_j = 5$ and $d_{ij} = 6$. So the conflict constraint prevents nodes i and j from appearing in the same subset.

Figure 6 Valid Cuts for Nodes Too Far Apart



The following statements use the MILP solver to compute each m_i, which has the more descriptive name *maxnodes[i]* here:

```
/* maxnodes[i] = max number of nodes among feasible subsets that contain node i */
num maxnodes {NODES};

/* UseNode[i] = 1 if node i is used, 0 otherwise */
var UseNode {NODES} binary;

max NumUsed = sum {i in NODES} UseNode[i];

con PopulationBounds:
   target - deviation_ub <= sum {i in NODES} p[i] * UseNode[i] <= target + deviation_ub;

problem MaxNumUsed include
   UseNode NumUsed PopulationBounds;
use problem MaxNumUsed;

put 'Finding max number of nodes among feasible subsets that contain node i...';
cofor {i in NODES} do;
   put i=;
   fix UseNode[i] = 1;
   solve with MILP / loglevel=0 logfreq=0;
   maxnodes[i] = round(NumUsed.sol);
   unfix UseNode;
end;
```

Again, the PROBLEM and USE PROBLEM statements control which variables, objective, and constraints declared in this PROC OPTMODEL session are active when you invoke the solver. As mentioned earlier, these problems are independent for each node, so you can use a COFOR loop to solve them concurrently. Within the body of the COFOR loop, you use the FIX and UNFIX statements to force **UseNode[i]** $= 1$ or not. When the loop terminates, all values of *maxnodes[i]* have been computed.

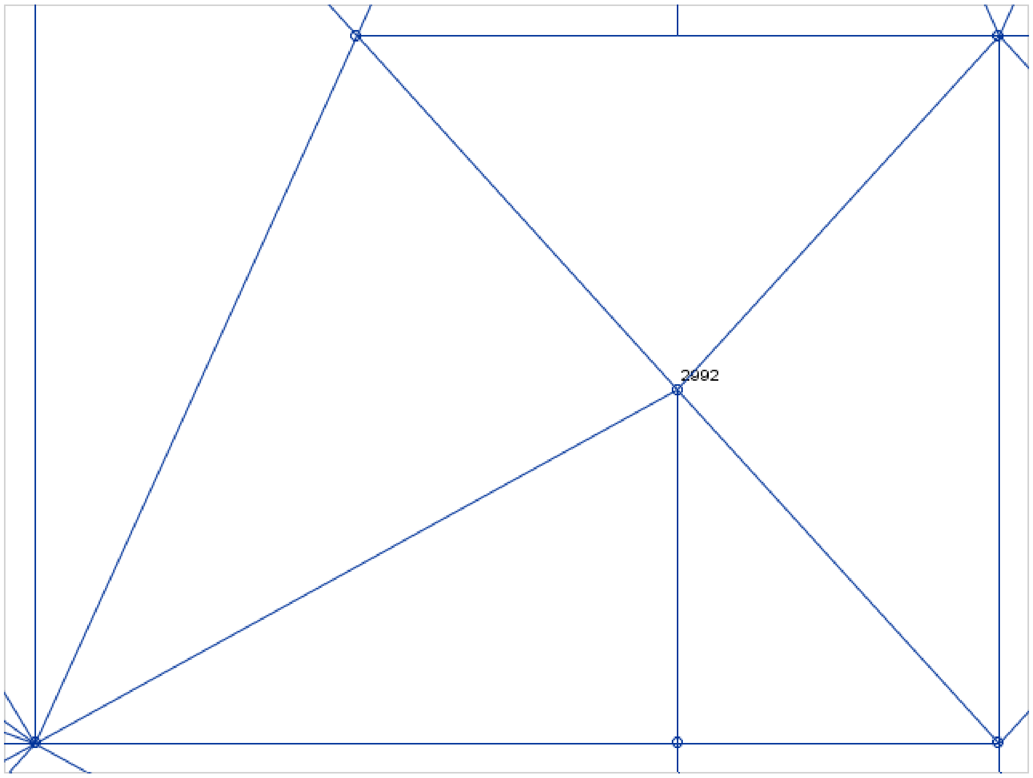
The following statements call the network solver and use the SHORTPATH option to find the shortest path distances d_{ij}, called *distance[i, j]* here:

```
/* distance[i,j] = shortest path distance (number of edges) from i to j */
num distance {NODES, NODES};
/* num one {EDGES} = 1; */
put 'Finding all-pairs shortest path distances...';
solve with NETWORK /
   links     = (include=EDGES /*weight=one*/)
   shortpath
   out       = (spweights=distance);
```

The edge weights default to 1, but you can also supply explicit edge weights by using the WEIGHT= option, as shown in the commented code.

Another way to encourage connectivity is to prevent small-population nodes from appearing without some neighbor in the moat around that node also appearing, as shown in Figure 7.

Figure 7 Valid Cuts for Small-Population Nodes

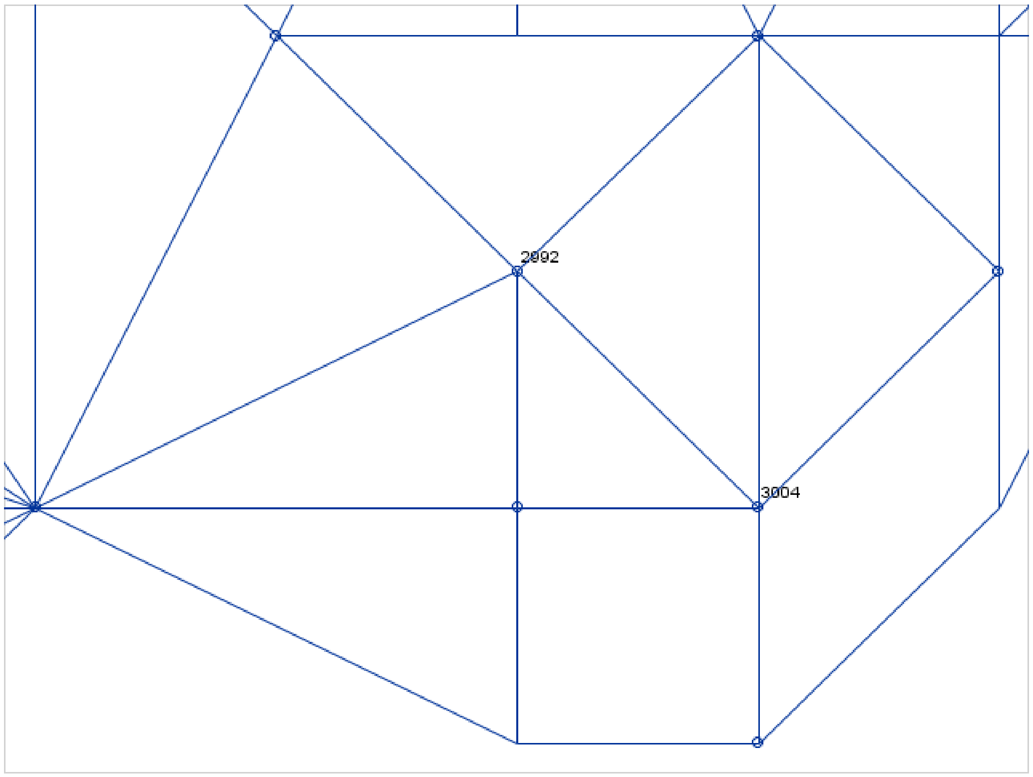


The rule is that if $x_i = 1$ and $p_i < \ell$, then $x_j = 1$ for some neighbor j of i. The following valid cut enforces that rule:

$$x_i \le \sum_{j \in N_i} x_j \tag{2}$$

A similar cut applies to pairs of small-population nodes, as shown in Figure 8.

Figure 8 Valid Cuts for Pairs of Small-Population Nodes



The rule is that if $x_i = x_j = 1$ and $p_i + p_j < \ell$, then $x_k = 1$ for some neighbor k in the moat around $\{i, j\}$. The following valid cut enforces that rule:

$$x_i + x_j - 1 \le \sum_{k \in (N_i \cup N_j) \setminus \{i,j\}} x_k$$

In fact, you can replace this cut with two stronger cuts:

$$x_i \le \sum_{k \in (N_i \cup N_j) \setminus \{i,j\}} x_k \tag{3}$$

$$x_j \le \sum_{k \in (N_i \cup N_j) \setminus \{i,j\}} x_k \tag{4}$$

This idea generalizes to small-population sets of more than two nodes, and the resulting cuts are known as "ring inequalities" in the literature (Martins, Constantino, and Borges 1999). Explicitly, the rule is that if $x_i = 1$ for some $i \in S \subset N$ such that $\sum_{j \in S} p_j < \ell$, then $x_k = 1$ for some neighbor k in the moat around S. The following valid cut enforces that rule:

$$x_i \le \sum_{k \in \left(\cup_{j \in S} N_j\right) \setminus S} x_k$$

The following statements declare the cuts in inequalities (1)–(4) and then declare a problem named Subproblem:

```
set NODE_PAIRS = {i in NODES, j in NODES: i < j};

/* cannot have two nodes i and j that are too far apart */
con Conflict {<i,j> in NODE_PAIRS: distance[i,j] >= min(maxnodes[i],maxnodes[j])}:
   UseNode[i] + UseNode[j] <= 1;
```

```
/* if UseNode[i] = 1 and p[i] < target - deviation_ub
   then UseNode[j] = 1 for some neighbor j of i */
con Moat1 {i in NODES: p[i] < target - deviation_ub}:
   UseNode[i] <= sum {j in NEIGHBORS[i]} UseNode[j];

/* if (UseNode[i] = 1 or UseNode[j] = 1) and p[i] + p[j] < target - deviation_ub
   then UseNode[k] = 1 for some neighbor k in moat around {i,j} */
con Moat2_i {<i,j> in NODE_PAIRS: p[i] + p[j] < target - deviation_ub}:
   UseNode[i] <= sum {k in (NEIGHBORS[i] union NEIGHBORS[j]) diff {i,j}} UseNode[k];
con Moat2_j {<i,j> in NODE_PAIRS: p[i] + p[j] < target - deviation_ub}:
   UseNode[j] <= sum {k in (NEIGHBORS[i] union NEIGHBORS[j]) diff {i,j}} UseNode[k];

problem Subproblem include
   UseNode PopulationBounds Conflict Moat1 Moat2_i Moat2_j;
```

Note that the **UseNode** variable and PopulationBounds constraint were already declared.

Calling the CLP Solver and Accessing Multiple Solutions

The following statements make Subproblem active and invoke the CLP solver with the FINDALLSOLNS option to find all solutions:

```
use problem Subproblem;
put 'Solving CLP subproblem...';
solve with CLP / findallsolns;
```

The automatically defined PROC OPTMODEL numeric parameter _NSOL_ contains the resulting number of solutions, and you can access the values of the variables in each solution *s* by using the `.SOL[s]` suffix:

```
/* use _NSOL_ and .sol[s] to access multiple solutions */
SUBSETS = 1.._NSOL_;
for {s in SUBSETS} NODES_s[s] = {i in NODES: UseNode[i].sol[s] > 0.5};
```

The log in Figure 9 shows that the CLP solver found 4,472 subsets that satisfy both the population range constraints and the cuts.

Figure 9 Constraint Programming Solver Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 66 variables (0 free, 0 fixed).
NOTE: The problem has 66 binary and 0 integer variables.
NOTE: The problem has 5123 linear constraints (5122 LE, 0 EQ, 0 GE, 1 range).
NOTE: The problem has 42926 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The experimental CLP solver is called.
NOTE: All possible solutions have been found.
NOTE: Number of solutions found = 4472.

CARD(SUBSETS)=4472
```

But some of these subsets might still be disconnected because the valid cuts only encourage and do not enforce connectivity.

Calling the Network Solver to Eliminate Disconnected Subsets

You can call the network solver with the CONCOMP option to find the connected components of the subgraph induced by each subset. If the subgraph is disconnected, remove it from consideration. These problems are independent, so you can again use a COFOR loop to solve them concurrently, as in the following statements:

```
/* check connectivity */
set NODES_THIS;
num component {NODES_THIS};
set DISCONNECTED init {};
cofor {s in SUBSETS} do;
   put s=;
   NODES_THIS = NODES_s[s];
   solve with NETWORK /
      links    = (include=EDGES)
      subgraph = (nodes=NODES_THIS)
      concomp
      out      = (concomp=component);
   if or {i in NODES_THIS} (component[i] > 1)
   then DISCONNECTED = DISCONNECTED union {s};
end;
SUBSETS = SUBSETS diff DISCONNECTED;
```

The log in Figure 10 shows that 2,771 of the 4,472 subsets are connected.

Figure 10 Network Solver Log

```
s=1
NOTE: The SUBGRAPH= option filtered 145 elements from 'EDGES.'
NOTE: The number of nodes in the input graph is 4.
NOTE: The number of links in the input graph is 3.
NOTE: Processing connected components.
NOTE: The graph has 1 connected component.
NOTE: Processing connected components used 0.00 (cpu: 0.00) seconds.
...
s=28
NOTE: The SUBGRAPH= option filtered 144 elements from 'EDGES.'
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 4.
NOTE: Processing connected components.
NOTE: The graph has 2 connected components.
NOTE: Processing connected components used 0.00 (cpu: 0.00) seconds.
...
s=4472
NOTE: The SUBGRAPH= option filtered 144 elements from 'EDGES.'
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 4.
NOTE: Processing connected components.
NOTE: The graph has 1 connected component.
NOTE: Processing connected components used 0.00 (cpu: 0.00) seconds.

CARD(SUBSETS)=2771
```

Because each network solver call finds the connected components so quickly in this example, the parallelization is not expected to speed up the performance. Nevertheless, the COFOR syntax makes it easy to try.

BACK TO THE MASTER MILP PROBLEM

Now that all feasible columns have been generated, the following statements solve the master MILP problem already declared:

```
use problem Master;
put 'Solving master problem, minimizing Objective1...';
solve;
```

The log in Figure 11 shows 2,771 binary variables (one per feasible column) and 66 + 1 (Partition and Cardinality) constraints, and the MILP solver solved the problem in two seconds.

Figure 11 MILP Solver Log

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2771 variables (0 free, 0 fixed).
NOTE: The problem has 2771 binary and 0 integer variables.
NOTE: The problem has 67 linear constraints (0 LE, 67 EQ, 0 GE, 0 range).
NOTE: The problem has 17334 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 1092 variables and 0 constraints.
NOTE: The MILP presolver removed 7089 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 1679 variables, 67 constraints, and 10245
      constraint coefficients.
NOTE: The MILP solver is called.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       0              .     96.3214286        .       1
...
             0       1       0              .    113.1320755        .       1
             0       1       1    124.0000000    117.1666667    5.83%       2
NOTE: The MILP solver added 40 cuts with 2752 cut coefficients at the root.
             4       0       1    124.0000000    124.0000000    0.00%       2
NOTE: Optimal.
NOTE: Objective = 124.
```

The total absolute deviation from the target population of 12,000 is 124. The feasible solution shown earlier in Figure 4 is actually the resulting optimal solution, and you can visually verify that it contains 14 connected clusters, each with population within the bounds of 12,000 $\pm$ 100.

An Alternative Objective

The previous MILP solver call minimizes the sum of absolute deviations (L_1 norm). The following statements declare an alternative objective and corresponding constraint to minimize the maximum deviation (L_∞ norm) instead:

```
/* L_infinity norm */
var MaxDeviation >= 0;
min Objective2 = MaxDeviation;
con MaxDeviationDef {s in SUBSETS}:
   MaxDeviation >= subset_deviation[s] * UseSubset[s];
```

You can specify the PRIMALIN option in the SOLVE WITH MILP statement to use the current solution as a starting point:

```
put 'Solving master problem, minimizing Objective2...';
MaxDeviation = max {s in SUBSETS} (subset_deviation[s] * UseSubset[s].sol);
solve with MILP / primalin;
```

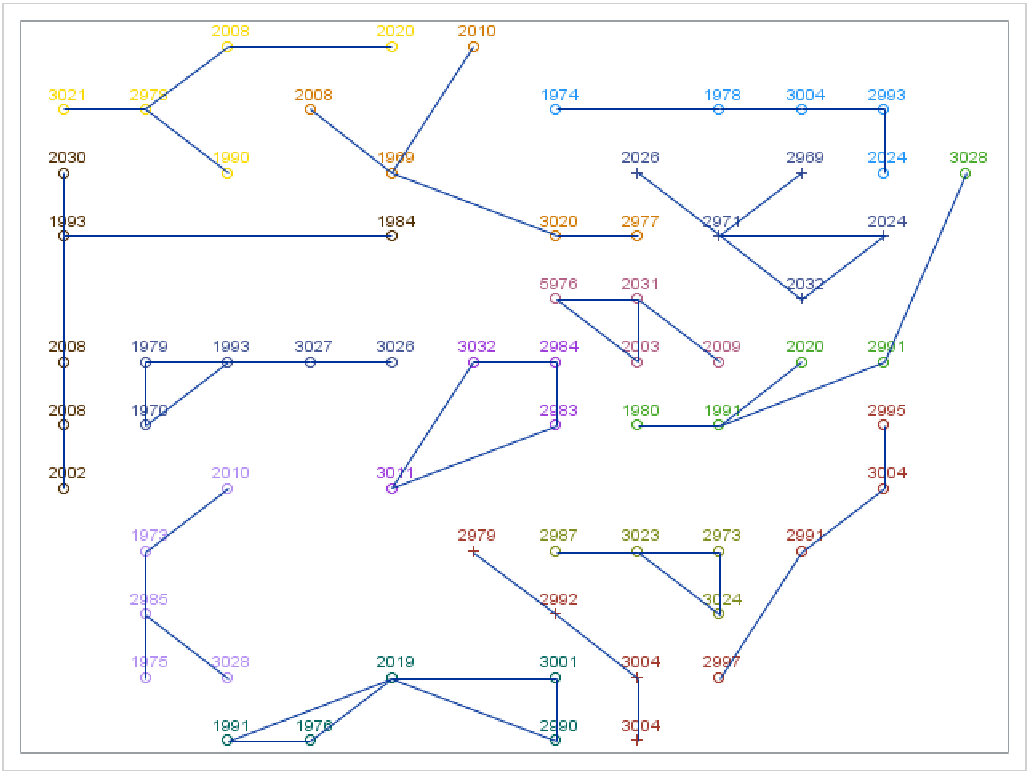
The log in Figure 12 shows that the starting solution has an objective value of 37 and the optimal solution has an objective value of 29.

Figure 12 MILP Solver Log

```
NOTE: The problem has 2772 variables (0 free, 0 fixed).
NOTE: The problem has 2771 binary and 0 integer variables.
NOTE: The problem has 2838 linear constraints (0 LE, 67 EQ, 2771 GE, 0 range).
NOTE: The problem has 22845 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 1092 variables and 1110 constraints.
NOTE: The MILP presolver removed 9278 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 1680 variables, 1728 constraints, and 13567
      constraint coefficients.
NOTE: The MILP solver is called.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       1     37.0000000      5.8468795  532.82%       1
...
             0       1       1     37.0000000     26.0503748   42.03%       4
NOTE: The MILP solver added 89 cuts with 4657 cut coefficients at the root.
             3       1       2     36.0000000     29.0000000   24.14%       5
             6       2       3     35.0000000     29.0000000   20.69%       5
             7       1       5     35.0000000     29.0000000   20.69%       6
            64      54       6     29.0000000     29.0000000    0.00%       7
NOTE: Optimal.
NOTE: Objective = 29.
```

The PROC SGPLOT output in Figure 13 shows the resulting optimal solution with respect to the second objective.

Figure 13 Optimal Solution for 14 Clusters, Bounds 12,000 ± 100, **Objective2**



The cluster in the lower left corner replaced a population-2,002 node with a population-2,010 node to increase the cluster's population from 12,000 – 37 to 12,000 – 29.

The entire process, including all data processing and solver calls, takes less than one minute.

CONCLUSION

This paper demonstrates the power and flexibility of the OPTMODEL procedure in SAS/OR to formulate and solve a complex optimization problem. The rich and expressive algebraic modeling language enables you to easily explore multiple interrelated mathematical programming formulations and access multiple optimization solvers, all within one PROC OPTMODEL session. The COFOR statement, new in SAS/OR 13.1, also offers a simple way to exploit parallel processing by solving independent problems concurrently.

REFERENCES

Carvajal, R., Constantino, M., Goycoolea, M., Vielma, J. P., and Weintraub, A. (2013). “Imposing Connectivity Constraints in Forest Planning Models.” *Operations Research* 61:824–836. `http://dx.doi.org/10.1287/opre.2013.1183`.

Martins, I., Constantino, M., and Borges, J. G. (1999). “Forest Management Models with Spatial Structure Constraints.” Working paper, University of Lisbon.

Shasha, D. E. (2002). *Doctor Ecco’s Cyberpuzzles: 36 Puzzles for Hackers and Other Mathematical Detectives*. New York: W. W. Norton.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Rob Pratt
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-1099
Rob.Pratt@sas.com