Paper SAS3161-2016

Using the OPTMODEL Procedure in SAS/OR® to Find the k Best Solutions

Rob Pratt, SAS Institute Inc.

ABSTRACT

Because optimization models often do not capture some important real-world complications, a collection of optimal or near-optimal solutions can be useful for decision makers. This paper uses various techniques for finding the k best solutions to the linear assignment problem in order to illustrate several features recently added to the OPTMODEL procedure in SAS/OR® software. These features include the network solver, the constraint programming solver (which can produce multiple solutions), and the COFOR statement (which allows parallel execution of independent solver calls).

WHY MULTIPLE SOLUTIONS?

Although many optimization problems have a unique optimal solution, there are several reasons to find multiple optimal or near-optimal solutions. One reason is that, because every model is a simplification of reality, some real-world complications are not captured; so it is useful to have a number of good solutions to compare with respect to any exogenous factors. Another motivation for multiple solutions is to provide a backup plan in case some input changes and there is not enough time to reoptimize. If your problem has competing objectives, a set of solutions can help you balance trade-offs among the objectives. A final reason to return multiple solutions for an optimization problem is to increase buy-in from stakeholders, who might feel more ownership if they select the final solution from a set of proposals.

LINEAR ASSIGNMENT PROBLEM

This paper uses various techniques for finding the k best solutions to the linear assignment problem to illustrate several features added to the OPTMODEL procedure in the past few releases of SAS/OR software. This classical optimization problem is to assign workers $1,\dots,n$ to jobs $1,\dots,n$ so that each worker is assigned to exactly one job and each job is assigned exactly one worker, while minimizing the total cost of the assignments. To express a mathematical programming formulation of the problem, you can first define a binary decision variable:

$$x_{ij} = \begin{cases} 1 & \text{if worker } i \text{ is assigned to job } j \\ 0 & \text{otherwise} \end{cases}$$

Let c_{ij} denote the cost of assigning worker i to job j. You can then define an objective and constraints as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} && (1)\\
\text{subject to} \quad & \sum_{j=1}^{n} x_{ij} = 1 \quad \text{for } i \in \{1,\dots,n\} && (2)\\
& \sum_{i=1}^{n} x_{ij} = 1 \quad \text{for } j \in \{1,\dots,n\} && (3)\\
& x_{ij} \in \{0,1\} \quad \text{for } i \in \{1,\dots,n\},\ j \in \{1,\dots,n\} && (4)
\end{aligned}$$

The objective (1) is to minimize the total cost of the assignments. Constraints (2) enforce the rule of exactly one job per worker, constraints (3) enforce the rule of exactly one worker per job, and constraints (4) force the decision variables to be binary. It turns out that the constraint matrix is totally unimodular, which implies that you can relax the integrality constraints (4) to $x_{ij} \geq 0$ and still automatically get an integer solution.

Figure 1 shows the cost matrix for an example instance from Murty (1968). In this matrix, each row corresponds to a worker, each column corresponds to a job, and the (i, j) entry is the cost c_{ij}.

Figure 1 Example Cost Matrix from Murty (1968)

$$\begin{pmatrix}
7 & 51 & 52 & 87 & 38 & 60 & 74 & 66 & 0 & 20 \\
50 & 12 & 0 & 64 & 8 & 53 & 0 & 46 & 76 & 42 \\
27 & 77 & 0 & 18 & 22 & 48 & 44 & 13 & 0 & 57 \\
62 & 0 & 3 & 8 & 5 & 6 & 14 & 0 & 26 & 39 \\
0 & 97 & 0 & 5 & 13 & 0 & 41 & 31 & 62 & 48 \\
79 & 68 & 0 & 0 & 15 & 12 & 17 & 47 & 35 & 43 \\
76 & 99 & 48 & 27 & 34 & 0 & 0 & 0 & 28 & 0 \\
0 & 20 & 9 & 27 & 46 & 15 & 84 & 19 & 3 & 24 \\
56 & 10 & 45 & 39 & 0 & 93 & 67 & 79 & 19 & 38 \\
27 & 0 & 39 & 53 & 46 & 24 & 69 & 46 & 23 & 1
\end{pmatrix}$$

The unique optimal solution, with an objective value of 0, for this instance turns out to be $x_{1,9} = x_{2,7} = x_{3,3} = x_{4,8} = x_{5,6} = x_{6,4} = x_{7,10} = x_{8,1} = x_{9,5} = x_{10,2} = 1$, with all other $x_{ij} = 0$. Figure 2 uses highlighting to show the entries that correspond to $x_{ij} = 1$ in the cost matrix. You can visually verify that this solution is feasible by noting that each row or column contains exactly one highlighted entry. Also, the lower bound of 0 for the objective value is clear because all the costs are nonnegative.

Figure 2 Optimal Solution to Example Instance

$$\begin{pmatrix}
7 & 51 & 52 & 87 & 38 & 60 & 74 & 66 & \boxed{0} & 20 \\
50 & 12 & 0 & 64 & 8 & 53 & \boxed{0} & 46 & 76 & 42 \\
27 & 77 & \boxed{0} & 18 & 22 & 48 & 44 & 13 & 0 & 57 \\
62 & 0 & 3 & 8 & 5 & 6 & 14 & \boxed{0} & 26 & 39 \\
0 & 97 & 0 & 5 & 13 & \boxed{0} & 41 & 31 & 62 & 48 \\
79 & 68 & 0 & \boxed{0} & 15 & 12 & 17 & 47 & 35 & 43 \\
76 & 99 & 48 & 27 & 34 & 0 & 0 & 0 & 28 & \boxed{0} \\
\boxed{0} & 20 & 9 & 27 & 46 & 15 & 84 & 19 & 3 & 24 \\
56 & 10 & 45 & 39 & \boxed{0} & 93 & 67 & 79 & 19 & 38 \\
27 & \boxed{0} & 39 & 53 & 46 & 24 & 69 & 46 & 23 & 1
\end{pmatrix}$$

The second-best solution, with an objective value of 1, turns out to be $x_{1,9} = x_{2,7} = x_{3,3} = x_{4,2} = x_{5,6} = x_{6,4} = x_{7,8} = x_{8,1} = x_{9,5} = x_{10,10} = 1$, with all other $x_{ij} = 0$. Figure 3 uses highlighting to show the entries that correspond to $x_{ij} = 1$.

Figure 3 Second-Best Solution to Example Instance

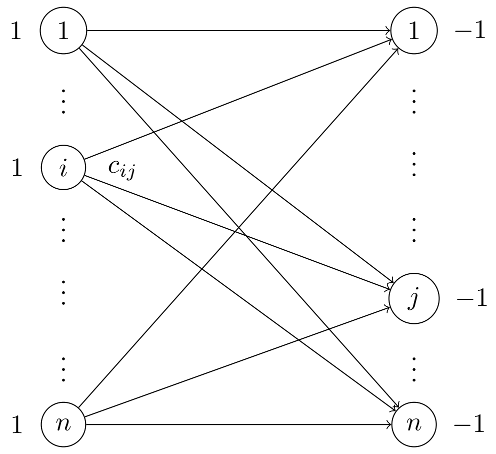
$$\begin{pmatrix}
7 & 51 & 52 & 87 & 38 & 60 & 74 & 66 & \boxed{0} & 20 \\
50 & 12 & 0 & 64 & 8 & 53 & \boxed{0} & 46 & 76 & 42 \\
27 & 77 & \boxed{0} & 18 & 22 & 48 & 44 & 13 & 0 & 57 \\
62 & \boxed{0} & 3 & 8 & 5 & 6 & 14 & 0 & 26 & 39 \\
0 & 97 & 0 & 5 & 13 & \boxed{0} & 41 & 31 & 62 & 48 \\
79 & 68 & 0 & \boxed{0} & 15 & 12 & 17 & 47 & 35 & 43 \\
76 & 99 & 48 & 27 & 34 & 0 & 0 & \boxed{0} & 28 & 0 \\
\boxed{0} & 20 & 9 & 27 & 46 & 15 & 84 & 19 & 3 & 24 \\
56 & 10 & 45 & 39 & \boxed{0} & 93 & 67 & 79 & 19 & 38 \\
27 & 0 & 39 & 53 & 46 & 24 & 69 & 46 & 23 & \boxed{1}
\end{pmatrix}$$

Later sections of this paper generalize from the two best solutions to the k best solutions to the linear assignment problem.

NETWORK MODEL

In addition to an explicit mathematical programming formulation, you can also model the linear assignment problem in terms of finding a minimum-cost flow in a bipartite network, as shown in Figure 4. The nodes on the left correspond to the workers, each with a supply of one unit. The nodes on the right correspond to the jobs, each with a demand of one unit. Directed arc (i, j) from worker i to job j has unit cost c_{ij}, and the goal is to find a minimum-cost feasible flow.

Figure 4 Network Model



SAS/OR provides 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 in PROC OPTMODEL via the SOLVE WITH NETWORK statement. Some of the network algorithms, such as connected components and maximal cliques, are diagnostic. Others solve classical network optimization problems, including minimum-cost network flow, the traveling salesman problem, and linear assignment. For more information, see the network solver chapter in *SAS/OR User's Guide: Mathematical Programming*.

PROC OPTNET

The following DATA step creates a dense data set that contains the example cost matrix shown earlier:

```
/* n = number of rows = number of columns */
%let n = 10;
data matrix;
   input c1-c&n;
   datalines;
 7 51 52 87 38 60 74 66  0 20
50 12  0 64  8 53  0 46 76 42
27 77  0 18 22 48 44 13  0 57
62  0  3  8  5  6 14  0 26 39
 0 97  0  5 13  0 41 31 62 48
79 68  0  0 15 12 17 47 35 43
76 99 48 27 34  0  0  0 28  0
 0 20  9 27 46 15 84 19  3 24
56 10 45 39  0 93 67 79 19 38
27  0 39 53 46 24 69 46 23  1
;
```

The simplest way to solve the corresponding linear assignment problem is to use the following PROC OPTNET statements. LAP is an alias for LINEAR_ASSIGNMENT.

```
/* find optimal solution by calling PROC OPTNET */
proc optnet data_matrix=matrix;
   lap out=out;
run;
```

PROC OPTNET also accepts sparse input with one observation per link, as shown in the following statements:

```
/* find optimal solution by calling PROC OPTNET with sparse input */
data links(keep=from to weight);
   array c[&n];
   set matrix;
   from = compress('row'||put(_N_,best.));
   do j = 1 to &n;
      to = compress('col'||put(j,best.));
      weight = c[j];
      if weight ne . then output;
   end;
run;
proc optnet data_links=links graph_direction=directed;
   lap out=out;
run;
```

This form of input is especially useful when a large portion of worker-job pairs are ineligible for assignment. You simply omit any such ineligible pairs from the DATA_LINKS= data set.

PROC OPTMODEL

The following PROC OPTMODEL statements use the network solver to solve the same problem by using the SOLVE WITH NETWORK statement:

```
/* find optimal solution by calling network solver in PROC OPTMODEL */
proc optmodel;
   num n = &n;
   set NSET = 1..n;
   num c {NSET, NSET};
   read data matrix into [_n_] {j in NSET} <c[_n_,j]=col('c'||j)>;

   set ARCS = setof {i in NSET, j in NSET} <'row'||i,'col'||j>;
   num cost {ARCS};
   for {i in NSET, j in NSET} cost['row'||i,'col'||j] = c[i,j];
   set <str,str> ASSIGNMENTS;

   /* call network solver */
   solve with network /
      LAP direction=directed links=(weight=cost) out=(assignments=ASSIGNMENTS);
   put ASSIGNMENTS=;
quit;
```

The log in Figure 5 shows that the network solver found the same optimal solution as in Figure 2.

Figure 5 Network Solver Log

```
NOTE: The number of nodes in the input graph is 20.
NOTE: The number of links in the input graph is 100.
NOTE: Processing the linear assignment problem.
NOTE: Objective = 0.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
ASSIGNMENTS={<'row1','col9'>,<'row2','col7'>,<'row3','col3'>,<'row4','col8'>,<
'row5','col6'>,<'row6','col4'>,<'row7','col10'>,<'row8','col1'>,<'row9','col5'>,
<'row10','col2'>}
```

MILP AND LP SOLVERS

The following PROC OPTMODEL statements instead use the mixed integer linear programming (MILP) and linear programming (LP) solvers to solve the problem, with the mathematical programming formulation expressed explicitly as in (1)–(4). The RELAXINT option relaxes integrality on the decision variables.

```
/* find optimal solution by calling MILP or LP solver in PROC OPTMODEL */
proc optmodel;
   num n = &n;
   set NSET = 1..n;
   num c {NSET, NSET};
   read data matrix into [_n_] {j in NSET} <c[_n_,j]=col('c'||j)>;

   /* X[i,j] = 1 iff i assigned to j */
   var X {NSET, NSET} binary;
   min Z = sum {i in NSET, j in NSET} c[i,j] * X[i,j];
   con Assign_i {i in NSET}:
      sum {j in NSET} X[i,j] = 1;
   con Assign_j {j in NSET}:
      sum {i in NSET} X[i,j] = 1;

   set SUPPORT = {i in NSET, j in NSET: X[i,j].sol > 0.5};

   /* call MILP solver */
   solve;
   put SUPPORT=;

   /* call LP solver */
   solve with LP relaxint;
   put SUPPORT=;
quit;
```

Figure 6 and Figure 7 show the logs from the MILP and LP solvers, respectively.

Figure 6 MILP Solver Log

```
NOTE: The problem has 100 variables (0 free, 0 fixed).
NOTE: The problem has 100 binary and 0 integer variables.
NOTE: The problem has 20 linear constraints (0 LE, 20 EQ, 0 GE, 0 range).
NOTE: The problem has 200 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 100 variables, 20 constraints, and 200
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active   Sols    BestInteger      BestBound      Gap    Time
             0       1      1              0              0    0.00%       0
             0       0      1              0              0    0.00%       0
NOTE: Optimal.
NOTE: Objective = 0.
SUPPORT={<1,9>,<2,7>,<3,3>,<4,8>,<5,6>,<6,4>,<7,10>,<8,1>,<9,5>,<10,2>}
```

Figure 7 LP Solver Log

```
NOTE: The integrality restriction was relaxed for 100 variables.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 100 variables, 20 constraints, and 200
      constraint coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                          Objective
      Phase Iteration        Value         Time
       D 1          1     0.000000E+00        0
       D 2          2     0.000000E+00        0
       P 2         17     0.000000E+00        0
NOTE: Optimal.
NOTE: Objective = 0.
NOTE: The Dual Simplex solve time is 0.00 seconds.
SUPPORT={<1,9>,<2,7>,<3,3>,<4,8>,<5,6>,<6,4>,<7,10>,<8,1>,<9,5>,<10,2>}
```

CONSTRAINT 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, you have more direct access through the 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*.

ONE OPTIMAL SOLUTION

The following PROC OPTMODEL statements use the same mathematical programming formulation (1)–(4) as already shown for the MILP solver but instead call the CLP solver to solve the problem:

```
/* find optimal solution by calling CLP solver in PROC OPTMODEL */
proc optmodel;
   num n = &n;
   set NSET = 1..n;
   num c {NSET, NSET};
   read data matrix into [_n_] {j in NSET} <c[_n_,j]=col('c'||j)>;

   /* X[i,j] = 1 iff i assigned to j */
   var X {NSET, NSET} binary;
   min Z = sum {i in NSET, j in NSET} c[i,j] * X[i,j];
   con Assign_i {i in NSET}:
      sum {j in NSET} X[i,j] = 1;
   con Assign_j {j in NSET}:
      sum {i in NSET} X[i,j] = 1;

   set SUPPORT = {i in NSET, j in NSET: X[i,j].sol > 0.5};

   solve with CLP;
   put SUPPORT=;
quit;
```

But this approach does not exploit the advantages of constraint programming. The following PROC OPTMODEL statements instead use two sets of integer variables (**Assign** and **Cost**), together with constraints that use the

ALLDIFF predicate to force each worker to be assigned to a different job and the ELEMENT predicate to enforce the desired relationship between **Assign[i]** and **Cost[i]**. Because these predicates are used, PROC OPTMODEL automatically calls the CLP solver.

```
/* find optimal solution by calling CLP solver using predicates in PROC OPTMODEL */
proc optmodel;
   num n = &n;
   set NSET = 1..n;
   num c {NSET, NSET};
   read data matrix into [_n_] {j in NSET} <c[_n_,j]=col('c'||j)>;

   /* Assign[i] = j iff i assigned to j */
   var Assign {NSET} >= 1 <= n integer;
   con AlldiffCon:
      alldiff(Assign);

   /* Cost[i] = c[i,Assign[i]] */
   var Cost {NSET} integer;
   con ElementCon {i in NSET}:
      element(Assign[i], {j in NSET} c[i,j], Cost[i]);
   min Z = sum {i in NSET} Cost[i];

   set SUPPORT = {i in NSET, j in NSET: Assign[i].sol = j};

   /* call CLP solver */
   solve;
   put SUPPORT=;
quit;
```

Figure 8 shows the log from the CLP solver.

Figure 8 CLP Solver Log

```
NOTE: The problem has 20 variables (10 free, 0 fixed).
NOTE: The problem has 0 binary and 20 integer variables.
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 11 predicate constraints.
NOTE: The OPTMODEL presolver is disabled for problems with predicates.
NOTE: Required number of solutions found (1).
NOTE: Minimal objective value found: 0.
SUPPORT={<1,9>,<2,7>,<3,3>,<4,8>,<5,6>,<6,4>,<7,10>,<8,1>,<9,5>,<10,2>}
```

k OPTIMAL SOLUTIONS

The following PROC OPTMODEL statements use the MAXSOLNS= option for the CLP solver to find up to k (in this case, four) optimal solutions. 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.

```
/* k = number of solutions desired */
%let k = 4;

/* find (up to) k optimal solutions by solving one CLP */
proc optmodel;
   num n = &n;
   set NSET = 1..n;
   num c {NSET, NSET};
   read data matrix into [_n_] {j in NSET} <c[_n_,j]=col('c'||j)>;
```

```
      /* Assign[i] = j iff i assigned to j */
      var Assign {NSET} >= 1 <= n integer;
      con AlldiffCon:
         alldiff(Assign);

      /* Cost[i] = c[i,Assign[i]] */
      var Cost {NSET} integer;
      con ElementCon {i in NSET}:
         element(Assign[i], {j in NSET} c[i,j], Cost[i]);
      min Z = sum {i in NSET} Cost[i];

      num k = &k;
      set SOLS = 1.._NSOL_;
      set SUPPORT_ALL = {i in NSET, j in NSET, s in SOLS: Assign[i].sol[s] = j};
      solve with CLP / maxsolns=(k);
      print Assign;
      create data soldata(drop=s) from [s]=SOLS
         cost=(sum {<i,j,(s)> in SUPPORT_ALL} c[i,j])
         {i in NSET} <col('row'||i)=Assign[i].sol[s]>;
   quit;
```

Because the optimal solution is unique for the example instance, the CLP solver returns only one solution, as shown in Figure 9.

Figure 9 CLP Solver Log, MAXSOLNS= Option

```
NOTE: The problem has 20 variables (10 free, 0 fixed).
NOTE: The problem has 0 binary and 20 integer variables.
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 11 predicate constraints.
NOTE: All possible solutions have been found.
NOTE: Number of solutions found = 1.
NOTE: Minimal objective value found: 0.
```

k BEST SOLUTIONS

To find the k best solutions (possibly with different objective values) instead of up to k optimal solutions (all with the same objective value), you need a different approach. The following PROC OPTMODEL statements first modify the previous formulation by appending an *s* index to each variable and constraint. The LEXICO predicate then forces the resulting k solutions to be distinct by specifying that they appear in strict lexicographic order.

```
   /* find k best solutions by solving one CLP */
   proc optmodel;
      num k = &k;
      set SOLS = 1..k;
      num n = &n;
      set NSET = 1..n;
      num c {NSET, NSET};
      read data matrix into [_n_] {j in NSET} <c[_n_,j]=col('c'||j)>;

      /* Assign[i,s] = j iff i assigned to j in solution s */
      var Assign {NSET, SOLS} >= 1 <= n integer;
      con AlldiffCon {s in SOLS}:
         alldiff({i in NSET} Assign[i,s]);

      /* Cost[i,s] = c[i,Assign[i,s]] */
      var Cost {NSET, SOLS} integer;
      con ElementCon {i in NSET, s in SOLS}:
         element(Assign[i,s], {j in NSET} c[i,j], Cost[i,s]);
      min Z = sum {i in NSET, s in SOLS} Cost[i,s];

      /* use LEXICO LT to get distinct solutions */
      con LexicoCon {s in SOLS diff {k}}:
         lexico({i in NSET} Assign[i,s] < {i in NSET} Assign[i,s+1]);

      solve;
      print Assign;
      create data soldata(drop=s) from [s]=SOLS
         cost=(sum {i in NSET} Cost[i,s])
         {i in NSET} <col('row'||i)=Assign[i,s]>;
   quit;
   proc print;
   run;
```

Figure 10 shows the CLP solver log, with a total objective value of 22.

Figure 10 CLP Solver Log, $k = 4$ Best Solutions

```
NOTE: The problem has 80 variables (40 free, 0 fixed).
NOTE: The problem has 0 binary and 80 integer variables.
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 47 predicate constraints.
NOTE: The OPTMODEL presolver is disabled for problems with predicates.
NOTE: Required number of solutions found (1).
NOTE: Minimal objective value found: 22.
```

The four best solutions turn out to have objective values of 0, 1, 10, and 11, as shown in Figure 11.

Figure 11 $k = 4$ Best Solutions from CLP Solver

Obs	cost	row1	row2	row3	row4	row5	row6	row7	row8	row9	row10
1	11	1	7	3	2	6	4	8	9	5	10
2	10	1	7	3	8	6	4	10	9	5	2
3	1	9	7	3	2	6	4	8	1	5	10
4	0	9	7	3	8	6	4	10	1	5	2

k BEST SOLUTIONS WITH THE MILP SOLVER

You can also use the MILP solver to find the k best solutions; this section contains two approaches. One approach uses additional indices and binary variables, together with linear constraints that force distinct solutions. The other approach calls the MILP solver in a loop, adding a constraint in each iteration to avoid repeating the solutions already found.

ONE SOLVER CALL

For two binary solutions, x^1 and x^2, to be distinct, they must differ in at least one component. The following logical propositions are all equivalent, where $\neg$, $\wedge$, and $\vee$ are the negation, conjunction, and disjunction operators, respectively:

$$
\begin{aligned}
x^1 &\neq x^2 \\
\neg \bigwedge_j \left(x_j^1 = x_j^2\right) \\
\bigvee_j \left(x_j^1 \neq x_j^2\right) \\
\bigvee_j \left(x_j^1 + x_j^2 = 1\right) && (5)
\end{aligned}
$$

You can enforce the disjunction (5) by introducing additional binary variables y_j and using linear constraints:

$$\sum_j y_j \geq 1 \qquad (6)$$

$$y_j \leq x_j^1 + x_j^2 \leq 2 - y_j \text{ for all } j \qquad (7)$$

Constraint (6) forces $y_j = 1$ for at least one j. Constraints (7) enforce the rule that $y_j = 1$ implies $x_j^1 + x_j^2 = 1$, whereas $y_j = 0$ implies only the redundant range constraint $0 \leq x_j^1 + x_j^2 \leq 2$. (For a systematic approach to convert logical propositions to linear constraints, see Raman and Grossmann (1991).) If the number of ones, $\sum_j x_j$, is constant for all feasible solutions (as in the linear assignment problem), you can strengthen constraint (6) by increasing the right-hand side from 1 to 2. For $k > 2$ distinct solutions, you need a set of such constraints for each pair $\{s_1, s_2\}$ of solutions. The following PROC OPTMODEL statements first modify the previous MILP formulation (1)–(4) by appending an *s* index to each variable and constraint. The **Y[i,j,s1,s2]** variable and associated linear constraints that correspond to (6)–(7) then force the resulting k solutions to be distinct.

```
/* find k best solutions by solving one MILP */
proc optmodel;
   num k = &k;
   set SOLS = 1..k;
   set SOLPAIRS = {s1 in SOLS, s2 in SOLS: s1 < s2};
   num n = &n;
   set NSET = 1..n;
   num c {NSET, NSET};
   read data matrix into [_n_] {j in NSET} <c[_n_,j]=col('c'||j)>;

   /* X[i,j,s] = 1 iff i assigned to j in solution s */
   var X {NSET, NSET, SOLS} binary;
   min Z = sum {i in NSET, j in NSET, s in SOLS} c[i,j] * X[i,j,s];
   con Assign_i {i in NSET, s in SOLS}:
      sum {j in NSET} X[i,j,s] = 1;
   con Assign_j {j in NSET, s in SOLS}:
      sum {i in NSET} X[i,j,s] = 1;

   /* Y[i,j,s1,s2] = 1 implies X[i,j,s1] ne X[i,j,s2] */
   var Y {NSET, NSET, SOLPAIRS} binary;
   /* solutions must differ in at least one component */
*   con AtLeastOneDiff {<s1,s2> in SOLPAIRS}:
      sum {i in NSET, j in NSET} Y[i,j,s1,s2] >= 1;
   /* all feasible solutions have the same number of ones,
      so can use stronger constraint */
   con AtLeastTwoDiff {<s1,s2> in SOLPAIRS}:
      sum {i in NSET, j in NSET} Y[i,j,s1,s2] >= 2;
   /* if Y[i,j,s1,s2] = 1 then not(X[i,j,s1] = 1 and X[i,j,s2] = 1) */
   con Ycon1 {i in NSET, j in NSET, <s1,s2> in SOLPAIRS}:
      X[i,j,s1] + X[i,j,s2] + Y[i,j,s1,s2] <= 2;
   /* if Y[i,j,s1,s2] = 1 then not(X[i,j,s1] = 0 and X[i,j,s2] = 0) */
   con Ycon0 {i in NSET, j in NSET, <s1,s2> in SOLPAIRS}:
      Y[i,j,s1,s2] <= X[i,j,s1] + X[i,j,s2];
   set SUPPORT_ALL = {i in NSET, j in NSET, s in SOLS: X[i,j,s].sol > 0.5};
   num assign {NSET, SOLS};

   solve with MILP / logfreq=1000;
   for {<i,j,s> in SUPPORT_ALL} assign[i,s] = j;
   put SUPPORT_ALL=;
   print assign;
   num cost {s in SOLS} = sum {i in NSET, j in NSET} c[i,j] * X[i,j,s].sol;
   print cost;
quit;
```

Figure 12 shows the MILP solver log, with a total objective value of 22, as in Figure 10.

Figure 12 MILP Solver Log, $k = 4$ Best Solutions

```
NOTE: The problem has 1000 variables (0 free, 0 fixed).
NOTE: The problem has 1000 binary and 0 integer variables.
NOTE: The problem has 1286 linear constraints (1200 LE, 80 EQ, 6 GE, 0 range).
NOTE: The problem has 5000 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 1000 variables, 1286 constraints, and 5000
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       2    500.0000000              0    500.0       0
             0       1       2    500.0000000      0.6666667 74900.0%       0
NOTE: The MILP solver's symmetry detection found 200 orbits. The largest orbit
      contains 6 variables.
             0       1       3     44.0000000      0.6666667 6500.00%       0
             0       1       4     43.0000000      0.6666667 6350.00%       0
            78      63       5     37.0000000      5.5000000  572.73%       1
           194      82       6     25.0000000      5.5000000  354.55%       2
           293     109       7     24.0000000      5.5000000  336.36%       2
           710     440       8     21.9999990      7.0952381  210.07%       3
          1000      66       8     21.9999990      9.6666667  127.59%       4
          1194       0       8     21.9999990     21.9999990    0.00%       4
NOTE: Optimal.
NOTE: Objective = 21.999999.
SUPPORT_ALL={<1,9,1>,<2,7,1>,<3,3,1>,<4,8,1>,<5,6,1>,<6,4,1>,<7,10,1>,<8,1,1>,<9
,5,1>,<10,2,1>,<1,1,2>,<2,7,2>,<3,3,2>,<4,8,2>,<5,6,2>,<6,4,2>,<7,10,2>,<8,9,2>,
<9,5,2>,<10,2,2>,<1,1,3>,<2,7,3>,<3,3,3>,<4,2,3>,<5,6,3>,<6,4,3>,<7,8,3>,<8,9,3>
,<9,5,3>,<10,10,3>,<1,9,4>,<2,7,4>,<3,3,4>,<4,2,4>,<5,6,4>,<6,4,4>,<7,8,4>,<8,1,
4>,<9,5,4>,<10,10,4>}
```

This approach does not scale well but is shown here for completeness. The numbers of variables and constraints grow quadratically with respect to k.

ROW GENERATION

You can also use row generation to generate multiple solutions for problems that have only binary variables, such as the linear assignment problem. The idea is to call the MILP solver in a loop, dynamically adding new constraints that cut off all previous solutions found. You want to force a vector x of binary decision variables to differ from a given binary solution vector $\hat{x}$. The following are equivalent:

$$\begin{aligned}
& x \neq \hat{x} \\
& \neg\left[\left(\bigwedge_{j:\hat{x}_j=1} x_j\right) \bigwedge \left(\bigwedge_{j:\hat{x}_j=0} \neg x_j\right)\right] \\
& \neg\left(\bigwedge_{j:\hat{x}_j=1} x_j\right) \bigvee \neg\left(\bigwedge_{j:\hat{x}_j=0} \neg x_j\right) \\
& \left(\bigvee_{j:\hat{x}_j=1} \neg x_j\right) \bigvee \left(\bigvee_{j:\hat{x}_j=0} x_j\right) \\
& \sum_{j:\hat{x}_j=1} (1 - x_j) + \sum_{j:\hat{x}_j=0} x_j \geq 1 \qquad (8)
\end{aligned}$$

If the number of ones, $\sum_j x_j$, is constant for all feasible solutions (as in the linear assignment problem), you can replace constraint (8) with any of the following three stronger constraints:

$$\begin{aligned}
\sum_{j:\hat{x}_j=1} (1 - x_j) \quad + \sum_{j:\hat{x}_j=0} x_j \quad &\geq 2 \qquad (9) \\
\sum_{j:\hat{x}_j=1} (1 - x_j) \quad &\geq 1 \qquad (10) \\
\sum_{j:\hat{x}_j=0} x_j \quad &\geq 1 \qquad (11)
\end{aligned}$$

The following PROC OPTMODEL statements implement the row-generation approach just described, using the sparse constraint (10) to exclude previous solutions:

```
/* find k best solutions by solving k MILPs (row generation) */
proc optmodel printlevel=0;
   num n = &n;
   set NSET = 1..n;
   num c {NSET, NSET};
   read data matrix into [_n_] {j in NSET} <c[_n_,j]=col('c'||j)>;

   /* X[i,j] = 1 iff i assigned to j */
   var X {NSET, NSET} binary;
   min Z = sum {i in NSET, j in NSET} c[i,j] * X[i,j];
   con Assign_i {i in NSET}:
      sum {j in NSET} X[i,j] = 1;
   con Assign_j {j in NSET}:
      sum {i in NSET} X[i,j] = 1;

   set SUPPORT = {i in NSET, j in NSET: X[i,j].sol > 0.5};

   num k = &k;
   num numsols init 0;
   set SOLS = 1..numsols;

   set <num,num> SUPPORT_s {SOLS};
*   con Exclude {s in SOLS}:
      sum {<i,j> in SUPPORT[s]} (1 - X[i,j])
    + sum {<i,j> in NSET cross NSET diff SUPPORT[s]} X[i,j] >= 1;
   /* all feasible solutions have the same number of ones,
      so can use stronger (and sparser) cut */
   con Exclude {s in SOLS}:
      sum {<i,j> in SUPPORT_s[s]} (1 - X[i,j]) >= 1;

   for {1..k} do;
      solve with MILP / loglevel=0;
      numsols = numsols + 1;
      SUPPORT_s[numsols] = {i in NSET, j in NSET: X[i,j].sol > 0.5};
      put _OBJ_.sol SUPPORT_s[numsols]=;
   end;
quit;
```

The body of the FOR loop calls the MILP solver and records the support of each new solution. Note that the loop contains no declaration statements. As the value of *numsols* changes, the SOLVE statement automatically updates the Exclude constraints. Figure 13 shows the log from this row-generation approach, with one additional linear constraint per MILP solver call.

Figure 13 Row-Generation Log

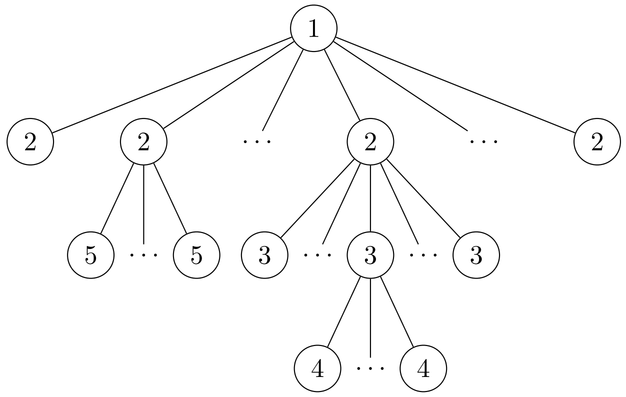
```
NOTE: The problem has 100 variables (0 free, 0 fixed).
NOTE: The problem has 100 binary and 0 integer variables.
NOTE: The problem has 20 linear constraints (0 LE, 20 EQ, 0 GE, 0 range).
NOTE: The problem has 200 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
0 SUPPORT_s[1]={<1,9>,<2,7>,<3,3>,<4,8>,<5,6>,<6,4>,<7,10>,<8,1>,<9,5>,<10,2>}
NOTE: The problem has 100 variables (0 free, 0 fixed).
NOTE: The problem has 100 binary and 0 integer variables.
NOTE: The problem has 21 linear constraints (0 LE, 20 EQ, 1 GE, 0 range).
NOTE: The problem has 210 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
1 SUPPORT_s[2]={<1,9>,<2,7>,<3,3>,<4,2>,<5,6>,<6,4>,<7,8>,<8,1>,<9,5>,<10,10>}
NOTE: The problem has 100 variables (0 free, 0 fixed).
NOTE: The problem has 100 binary and 0 integer variables.
NOTE: The problem has 22 linear constraints (0 LE, 20 EQ, 2 GE, 0 range).
NOTE: The problem has 220 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
10 SUPPORT_s[3]={<1,1>,<2,7>,<3,3>,<4,8>,<5,6>,<6,4>,<7,10>,<8,9>,<9,5>,<10,2>}
NOTE: The problem has 100 variables (0 free, 0 fixed).
NOTE: The problem has 100 binary and 0 integer variables.
NOTE: The problem has 23 linear constraints (0 LE, 20 EQ, 3 GE, 0 range).
NOTE: The problem has 230 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
11 SUPPORT_s[4]={<1,1>,<2,7>,<3,3>,<4,2>,<5,6>,<6,4>,<7,8>,<8,9>,<9,5>,<10,10>}
```

MURTY'S ALGORITHM

As discussed in previous sections, the network solver provides a specialized algorithm that solves the linear assignment problem but returns only one solution. Murty (1968) describes an algorithm to find the k best solutions to the linear assignment problem by solving $O(kn)$ linear assignment problems whose magnitude is the same as or smaller than that of the original problem. The key idea is that to find the next-best solution, you can solve at most $n-1$ linear assignment problems, so by iterating you can find the k best solutions. Murty's algorithm is implemented as a tree search in which the root node is the original linear assignment problem. Each node of the tree has sets of variables fixed to 0 or 1. Because the decision variables are only implicit in the network solver, you fix variables by modifying the input network. Explicitly, to fix variable x_{ij} to 0, remove arc (i,j); to fix variable x_{ij} to 1, remove all other arcs from i or to j. Because the algorithm removes arcs, each node in the search tree is a sparser linear assignment problem than its parent node. After solving a given node, you create at most $n-1$ child nodes by conditioning on the first difference from the optimal solution at that node.

Figure 14 shows one possible evolution of the tree, where the numbers indicate the order in which the nodes are created. Initially, the tree consists of only one node; this root node (labeled 1 in the figure) is solved, and the solution is recorded as the best. Next, $n-1$ child nodes (labeled 2) are created and solved, the best solution among these is recorded as the second-best solution overall, and the corresponding node becomes the next node on which to branch. Then $n-1$ new child nodes (labeled 3) are created and solved, and the best solution among all unbranched nodes (both 2 and 3) becomes the third-best solution overall. Branching on the corresponding node now yields $n-1$ additional children (labeled 4). The best solution among all unbranched nodes might now be a node labeled 2, and the next set of $n-1$ children (labeled 5) arises from there.

Figure 14 Branching Scheme



Because the $n-1$ child nodes that are created at each stage correspond to independent linear assignment problems, Murty's algorithm can potentially benefit from 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. In SAS/OR 14.1, the COFOR statement also supports running in a distributed environment across multiple machines.

The first several PROC OPTMODEL statements declare index sets and parameters and read the input data, as in previous examples:

```
/* find k best solutions by solving multiple LAPs,
   with concurrent (n-1)-way branching (Murty 1968) */
proc optmodel printlevel=0 prefertkfunc=never;
   num n = &n;
   num k = &k;
   num numsols init 0;
   set SOLS = 1..numsols;
   set <num,num> SUPPORT_s {SOLS};
   set NSET = 1..n;
   num c {NSET, NSET};
   read data matrix into [_n_] {j in NSET} <c[_n_,j]=col('c'||j)>;

   set ARCS = setof {i in NSET, j in NSET} <'row'||i,'col'||j>;
   num cost {ARCS};
   for {i in NSET, j in NSET} cost['row'||i,'col'||j] = c[i,j];
   set <str,str> ASSIGNMENTS;
```

The following statements declare additional index sets and parameters that are needed to implement Murty's algorithm. The set NODES contains all nodes in the tree and is partitioned into three disjoint subsets:

- UNSOLVED contains the nodes that have not yet been solved.
- CANDIDATES contains the nodes that have been solved but not declared to be among the best solutions found.

- WINNERS contains the nodes that will be returned by Murty's algorithm, which terminates when the cardinality of WINNERS is k.

```
num numnodes init 1;
set NODES = 1..numnodes; /* root is node 1 */
/* FIX[node,v] = for given node, set of <i,j> pairs with X[i,j] fixed to
   value v in {0,1} */
set <num,num> FIX {NODES, 0..1} init {};
set <num,num> SUPPORT {NODES} init {};
num obj {NODES} init .;
set UNSOLVED   init NODES;
set CANDIDATES init {};
set WINNERS    init {};
num candidate_obj {node in CANDIDATES} = obj[node];
num min_obj = min(of candidate_obj[*]);
num argmin_obj;

num node_this init 1;
/* fixing to 0 or 1 is equivalent to removing arcs */
set <str,str> ARCS_THIS =
   (ARCS diff setof {<i,j> in FIX[node_this,0]} <'row'||i,'col'||j>)
   diff setof {<i,j> in FIX[node_this,1], i2 in NSET, j2 in NSET: (i = i2) + (j = j2) = 1}
      <'row'||i2,'col'||j2>;
```

The following nested loops form the bulk of the algorithm. The outer FOR loop iterates k times (one iteration per desired solution), and the inner COFOR loop solves the linear assignment problem at all unsolved nodes concurrently.

```
/* each iteration of outer loop yields next best solution */
for {1..k} do;
   /* solve all unsolved nodes concurrently */
   cofor {node in UNSOLVED} do;
      node_this = node;
      put node_this=;
      for {v in 0..1} put FIX[node_this,v]=;
      solve with network /
         LAP direction=directed links=(weight=cost)
         subgraph=(links=ARCS_THIS) out=(assignments=ASSIGNMENTS);
      if card(ASSIGNMENTS) < n then obj[node_this] = .;
      else do;
         obj[node_this] = _OROPTMODEL_num_['OBJECTIVE'];
         SUPPORT[node_this] = {i in NSET, j in NSET: <'row'||i,'col'||j> in ASSIGNMENTS};
      end;
      put obj[node_this] SUPPORT[node_this];
      put ' ';
   end;
   CANDIDATES = CANDIDATES union {node in UNSOLVED: obj[node] ne .};
   UNSOLVED   = {};
   /* find candidate node with best objective */
   for {node in CANDIDATES: obj[node] = min_obj} do;
      argmin_obj = node;
      leave;
   end;
   put min_obj= argmin_obj=;
   put ' ';
   /* record this as next solution */
   numsols = numsols + 1;
   SUPPORT_s[numsols] = SUPPORT[argmin_obj];
   WINNERS            = WINNERS    union {argmin_obj};
   CANDIDATES         = CANDIDATES diff  {argmin_obj};
   /* if supply of good solutions is adequate, can stop branching */
   if numsols = k or
      (card(CANDIDATES) >= k-numsols
      and smallest(k-numsols, of candidate_obj[*]) <= obj[argmin_obj])
```

```
         then continue;
         /* create n - 1 new nodes by branching a la Murty
            (condition on first difference) */
         /* r = first difference (cannot be n if square instance) */
         for {r in 1..n-1} do;
            numnodes = numnodes + 1;
            UNSOLVED = UNSOLVED union {numnodes};
            FIX[numnodes,0] = FIX[argmin_obj,0]
               union setof {i in {r},    <(i),j> in SUPPORT[argmin_obj]} <i,j>;
            FIX[numnodes,1] = FIX[argmin_obj,1]
               union setof {i in 1..r-1, <(i),j> in SUPPORT[argmin_obj]} <i,j>;
            /* prune if trivially infeasible */
            if card(FIX[numnodes,0] inter FIX[numnodes,1]) > 0 then do;
               UNSOLVED = UNSOLVED diff {numnodes};
               numnodes = numnodes - 1;
            end;
         end;
      end;
```

After the outer FOR loop terminates, the following statements report the k best solutions to the log and terminate the procedure:

```
      put WINNERS=;
      for {node in WINNERS} put obj[node] SUPPORT[node];
   quit;
```

Figure 15 shows this portion of the log from Murty's algorithm, with the same four solutions as in Figure 11.

Figure 15 Murty's Algorithm Log

```
WINNERS={1,5,2,20}
0 {<1,9>,<2,7>,<3,3>,<4,8>,<5,6>,<6,4>,<7,10>,<8,1>,<9,5>,<10,2>}
1 {<1,9>,<2,7>,<3,3>,<4,2>,<5,6>,<6,4>,<7,8>,<8,1>,<9,5>,<10,10>}
10 {<1,1>,<2,7>,<3,3>,<4,8>,<5,6>,<6,4>,<7,10>,<8,9>,<9,5>,<10,2>}
11 {<1,1>,<2,7>,<3,3>,<4,2>,<5,6>,<6,4>,<7,8>,<8,9>,<9,5>,<10,10>}
```

Murty's algorithm tends to perform the best among all these approaches, as expected because it calls a specialized linear assignment algorithm and exploits parallelism.

CONCLUSION

This paper demonstrates the power and flexibility of the OPTMODEL procedure in SAS/OR to find multiple solutions for mathematical optimization problems, with linear assignment as an illustrative example. The rich and expressive algebraic modeling language enables you to easily formulate problems and access multiple solvers. You can also use the programming language provided by PROC OPTMODEL to write customized algorithms that call the solvers as subroutines. The COFOR statement offers a simple way to exploit parallel processing by solving independent problems concurrently, on either one machine or a grid.

REFERENCES

Murty, K. G. (1968). "An Algorithm for Ranking All the Assignments in Order of Increasing Cost." *Operations Research* 16:682–687. Letter to the editor. http://dx.doi.org/10.1287/opre.16.3.682.

Raman, R., and Grossmann, I. E. (1991). "Relation between MILP Modelling and Logical Inference for Chemical Process Synthesis." *Computers and Chemical Engineering* 15:73–84. http://dx.doi.org/10.1016/0098-1354(91)87007-V.

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