Paper 200-2011

Linear Optimization in SAS/OR® Software: Migrating to the OPTMODEL Procedure

Rob Pratt and Ed Hughes, SAS Institute Inc., Cary NC

ABSTRACT

PROC OPTMODEL, the flagship SAS/OR® optimization procedure, is intended to supersede the INTPOINT, LP, and NETFLOW procedures for linear optimization. PROC OPTMODEL's rich and flexible syntax enables natural and compact algebraic formulations of optimization problems. And the linear programming, mixed-integer linear programming, and network solvers available in PROC OPTMODEL are much faster than those in the INTPOINT, LP, and NETFLOW procedures.

Beyond the easier modeling and access to improved algorithms, PROC OPTMODEL also provides programming capabilities that enable you to develop customized solution methods. This paper uses several examples to illustrate how to migrate to PROC OPTMODEL.

INTRODUCTION

An optimization problem involves minimizing (or maximizing) an objective function subject to a set of constraints. The objective and constraint functions are expressed in terms of decision variables. Mathematically, a general optimization problem is described as

$$\begin{array}{ll} \text{minimize} \mid \text{maximize} & f(x) \\ \text{subject to} & g(x)\{\leq, =, \geq\}b \\ & \ell \leq x \leq u \end{array}$$

where x is the vector of decision variables, $f(x)$ is the objective function, $g(x)\{\leq, =, \geq\}b$ are the general constraints, and $\ell \leq x \leq u$ are the bound constraints (possibly infinite). If at least one of the functions in $f(x)$ or $g(x)$ is nonlinear, the problem is a *nonlinear programming* (NLP) problem. Otherwise, the problem is a *linear programming* (LP) problem. If some of the decision variables in an LP problem are required to take integer values, then the problem is instead called a *mixed-integer linear programming* (MILP) problem. A *linear optimization* (LP or MILP) problem can be expressed as

$$\begin{array}{lll} \text{minimize} \mid \text{maximize} & c^T x & \\ \text{subject to} & Ax\{\leq, =, \geq\}b & \\ & \ell \leq x \leq u & \\ & x_i \in \mathbb{Z} & \text{for } i \in I \end{array}$$

where

- $x \in \mathbb{R}^n$ is the vector of decision variables
- $c \in \mathbb{R}^n$ is the vector of objective function coefficients
- $A \in \mathbb{R}^{m \times n}$ is the matrix of constraint coefficients
- $b \in \mathbb{R}^m$ is the vector of constraint right-hand sides (RHS)
- $\ell \in \mathbb{R}^n$ is the vector of lower bounds on variables (possibly $-\infty$)
- $u \in \mathbb{R}^n$ is the vector of upper bounds on variables (possibly ∞)
- $I \subseteq \{1, \ldots, n\}$ is the index set of the integer variables (LP is the special case where I is the empty set)

If the constraint matrix A is the node-arc incidence matrix of a network, then the problem is called a *network optimization* problem.

The LP, INTPOINT, NETFLOW, and NLP procedures and their corresponding optimization solvers were developed several years before the introduction of the newer OPT* family of SAS/OR procedures and their linear, mixed-integer linear, quadratic, and general nonlinear optimization solvers. These newer optimization procedures and solvers deliver significant improvements over the older procedures and solvers in several areas:

- clarity and flexibility in optimization modeling, including more versatile use of input data
- flexibility in tailoring the solution process to the model, synthesizing standard and customized optimization methods as needed
- scalability and speed in the solvers

LINEAR OPTIMIZATION SOLVERS IN PROC OPTMODEL

LP SOLVER

The LP solver in PROC OPTMODEL includes four algorithms:

- primal simplex
- dual simplex
- network simplex (new in SAS/OR 9.3)
- interior point

These same algorithms are available through PROC OPTLP. The primal and dual simplex solvers implement the two-phase simplex method. In phase I, the solver tries to find a feasible solution. If no feasible solution is found, the LP is infeasible; otherwise, the solver enters phase II to solve the original LP. The network simplex solver extracts a network substructure, solves this using network simplex, and then constructs an advanced basis to feed to either primal or dual simplex. The interior point solver implements a primal-dual predictor-corrector interior point algorithm.

MILP SOLVER

The MILP solver in PROC OPTMODEL is a branch-and-bound algorithm based on linear programming. This divide-and-conquer approach attempts to solve the original problem by solving linear programming relaxations of a sequence of smaller subproblems and includes several features that are not present in PROC LP:

- advanced presolve techniques to reduce the problem size and improve numerical stability
- dynamically generated cutting planes to strengthen the LP relaxation
- primal heuristics to find integer feasible solutions

The same MILP solver is available through PROC OPTMILP.

OVERVIEW OF MIGRATION

For the reasons outlined earlier, the focus of research and development for mathematical optimization in SAS/OR software has shifted to the newer procedures and solvers. The LP, INTPOINT, NETFLOW, and NLP procedures are no longer under active development. But these legacy procedures are still fully supported, and in SAS/OR 9.3 they will be documented in their own book, *SAS/OR 9.3 User's Guide: Mathematical Programming Legacy Procedures*. At some point in the near future, support for the legacy procedures will be discontinued. Accordingly, all SAS/OR users are urged to plan now to migrate from these older procedures to the corresponding newer SAS/OR optimization procedures:

- for linear programming, from the LP procedure to the OPTLP or OPTMODEL procedure
- for mixed-integer linear programming, from the LP procedure to the OPTMILP or OPTMODEL procedure
- from the INTPOINT procedure to the OPTLP or OPTMODEL procedure
- from the NETFLOW procedure to the OPTLP or OPTMODEL procedure
- from the NLP procedure to the OPTMODEL procedure

Migration from PROC NLP to PROC OPTMODEL for nonlinear optimization is described in an earlier SAS Global Forum paper by Huang and Hughes (2010). Much of the material in that paper still applies here, but this paper focuses on linear optimization, including linear programming, mixed-integer linear programming, and network flows. For linear optimization, you can migrate to PROC OPTLP or PROC OPTMILP, but the recommended approach is to use PROC OPTMODEL. Both approaches are described in the following sections.

MIGRATION PATHS USING THE MPSOUT= OPTION

The LP, INTPOINT, and NETFLOW procedures have recently added the MPSOUT= option, which bypasses the procedure's solver and produces a SAS data set in a mathematical programming system (MPS) format that describes the problem that was to be solved with the older procedure. This data set can be supplied as an input data set to the OPTLP or OPTMILP procedure.

USING THE MPSOUT= OPTION TO MIGRATE FROM PROC INTPOINT TO PROC OPTLP

Suppose Node and Arc are SAS data sets that contain input for PROC INTPOINT. The following SAS statements read the input data, call the legacy interior point LP solver, and write the solution to a SAS data set called Solution:

```
/* original PROC INTPOINT code */
proc intpoint
   bytes     = 1000000
   nodedata  = Node
   arcdata   = Arc
   conout    = Solution;
run;
```

To try the new solvers, you can first use the following SAS statements, which read the same input data, skip the solver call, and instead create an MPS-format SAS data set called Mpsdata:

```
/* use MPSOUT= instead to create Mpsdata data set */
proc intpoint
   bytes    = 1000000
   nodedata = Node
   arcdata  = Arc
   mpsout   = Mpsdata;
run;
```

Then you can use the Mpsdata SAS data set as input to PROC OPTLP as follows:

```
/* call PROC OPTLP to use new solvers */
proc optlp data=Mpsdata primalout=Solution;
run;
```

Output 1 shows the output from the OPTLP call, which uses the default LP solver, dual simplex.

Output 1 Calling PROC OPTLP with the MPSOUT= Data Set Using Dual Simplex

```
                    The OPTLP Procedure

                      Problem Summary

Problem Name                          modname
Objective Sense                  Minimization
Objective Function                      objfn
RHS

Number of Variables                       64
Bounded Above                              0
Bounded Below                             12
Bounded Above and Below                   52
Free                                       0
Fixed                                      0

Number of Constraints                     20
LE (<=)                                    4
EQ (=)                                    16
GE (>=)                                    0
Range                                      0

Constraint Coefficients                  128

                    Solution Summary

  Solver                        Dual simplex
  Objective Function                  objfn
  Solution Status                   Optimal
  Objective Value                -1281110.35

  Primal Infeasibility                     0
  Dual Infeasibility                       0
  Bound Infeasibility                      0

  Iterations                              30
  Presolve Time                         0.00
  Solution Time                         0.00
```

The following three OPTLP calls use primal simplex, the new interior point solver, and network simplex, respectively. Of course, each solver finds the same optimal objective value, although the solutions found might differ if the problem

has alternative optimal solutions.

```
proc optlp data=Mpsdata solver=ps primalout=Solution_ps;
run;

proc optlp data=Mpsdata solver=ii primalout=Solution_ii;
run;

proc optlp data=Mpsdata solver=ns primalout=Solution_ns;
run;
```

USING THE MPSOUT= OPTION TO MIGRATE FROM PROC NETFLOW TO PROC OPTLP

For PROC NETFLOW, the required syntax is almost identical to PROC INTPOINT:

```
proc netflow
   bytes    = 1000000
   nodedata = Node
   arcdata  = Arc
   mpsout   = Mpsdata;
run;
quit;
```

As before, you can then use the resulting Mpsdata SAS data set as input to PROC OPTLP.

USING THE MPSOUT= OPTION TO MIGRATE FROM PROC LP TO PROC OPTLP OR PROC OPTMILP

LP Problem

Suppose Lpdata is a SAS data set that contains PROC LP input to solve an LP problem. The following SAS statements read the input data, call the legacy LP solver, and write the solution to a SAS data set called Solution:

```
/* original PROC LP code */
proc lp data=Lpdata primalout=Solution;
run;
```

To try the new solvers, you can first use the following SAS statements, which read the same input data, skip the solver call, and instead create an MPS-format SAS data set called Mpsdata:

```
/* use MPSOUT= instead to create Mpsdata data set */
proc lp data=Lpdata mpsout=Mpsdata;
run;
```

Then you can use the Mpsdata SAS data set as input to PROC OPTLP as follows:

```
/* call PROC OPTLP to use new solvers */
proc optlp data=Mpsdata primalout=Solution_optlp;
run;
```

MILP Problem

If the original PROC LP statements instead use a SAS data set called Milpdata as input to solve an MILP problem, the two steps to try the new solver are similar:

```
/* original PROC LP code */
proc lp data=Milpdata primalout=Solution;
run;

/* use MPSOUT= instead to create Mpsdata data set */
proc lp data=Milpdata mpsout=Mpsdata;
run;

/* call PROC OPTMILP to use new solver */
proc optmilp data=Mpsdata primalout=Solution_optmilp;
run;
```

USING PROC OPTMODEL TO FORMULATE AND SOLVE LINEAR OPTIMIZATION PROBLEMS

By writing just a few additional statements beyond what is needed to run one of the legacy procedures, you can use the MPSOUT= option described in the previous section to try out the new solvers. But a preferable approach is to model the problem using the algebraic modeling capabilities of PROC OPTMODEL, thus producing highly readable SAS statements that transparently depict the algebraic structure of the optimization model. For a full exploration of the rich syntax and vast number of statements and expressions of PROC OPTMODEL, see *SAS/OR User's Guide: Mathematical Programming*.

This section first describes the statements and expressions that are essential to formulate an optimization problem. Then two examples demonstrate how to model linear optimization problems with PROC OPTMODEL.

Table 1 shows the PROC OPTMODEL declaration statements used to define the elements of an optimization problem.

Table 1 Declaration Statements in PROC OPTMODEL

Statement	Description
NUMBER	Numeric parameter
STRING	String parameter
SET	Set whose members can be used for indexing constants, variables, or constraints
VAR	Decision variable
IMPVAR	Implicit variable
MIN | MAX	Objective function of a minimization or maximization problem
CONSTRAINT	Linear or nonlinear constraint

Alternative keywords for NUMBER, STRING, and CONSTRAINT are NUM, STR, and CON, respectively.

In an optimization problem, the mathematical expressions of the objective function or constraints frequently contain summation or product operations, such as

$$\sum_{i=1}^{n} a_i x_i \qquad \text{or} \qquad \prod_{i=1}^{n} y_i$$

For the INTPOINT, LP, and NETFLOW procedures, you must use DATA step DO loops for summation or product operations. Although PROC OPTMODEL also supports DO loops, the following SUM and PROD expressions in PROC OPTMODEL offer more natural ways to accomplish these tasks:

```
sum {i in 1..n} a[i] * x[i]
prod {i in 1..n} y[i]
```

PROC OPTMODEL also provides far better means for using external data than the legacy optimization procedures. The READ DATA statement in PROC OPTMODEL reads a SAS data set and has the following features:

- Data can be imported from multiple data sets.
- Data are imported from data sets as matrices or one-dimensional arrays, allowing transparent access of the data.
- Data can be input in sparse matrix format.

The CREATE DATA statement in PROC OPTMODEL allows this same flexibility in writing output data sets, instead of requiring you to use the DATA step or the SQL procedure to manipulate output data sets created by the legacy procedures.

MINIMUM COST NETWORK FLOW

Suppose you have a directed network (N, A), with node set N and arc set A. Each node $i \in N$ has supply or demand b_i. Each arc $(i, j) \in A$ has cost c_{ij} per unit of flow, lower bound ℓ_{ij}, and upper bound u_{ij}. The *minimum cost network flow problem* is to satisfy all supplies and demands at a minimum cost, while respecting all arc bounds.

The minimum cost network flow problem can be formulated as an LP problem, where x_{ij} is the amount of flow across arc (i, j):

$$\begin{aligned} \text{minimize} \quad & \sum_{(i,j)\in A} c_{ij}x_{ij} \\ \text{subject to} \quad & \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = b_i \quad \text{for } i \in N \\ & \ell_{ij} \le x_{ij} \le u_{ij} \quad \text{for } (i,j) \in A \end{aligned}$$

The Node and Arc input data sets (not displayed) are the same ones used for the PROC INTPOINT MPSOUT= example shown earlier. The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called Solution. Note that the model is completely separated from the data; you can solve any minimum cost network flow problem without changing the PROC OPTMODEL statements.

```
proc optmodel;
   set <str> NODES;
   num _supdem_ {NODES} init 0;
   read data Node into NODES=[_node_] _supdem_;

   set <str,str> ARCS;
   num _lo_ {ARCS} init 0;
   num _capac_ {ARCS} init .;
   num _cost_ {ARCS};

   read data Arc nomiss into ARCS=[_tail_ _head_] _lo_ _capac_ _cost_;
   NODES = NODES union (union {<i,j> in ARCS} {i,j});

   var Flow {<i,j> in ARCS} >= _lo_[i,j];
   for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
   min TotalCost = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
   con balance {i in NODES}:
      sum {<(i),j> in ARCS} Flow[i,j] - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];

   num infinity = min {r in {}} r;
   num excess = sum {i in NODES} _supdem_[i];
   if (excess > 0) then do;
      /* change equality constraint to le constraint */
      for {i in NODES: _supdem_[i] > 0} balance[i].lb = -infinity;
   end;
   else if (excess < 0) then do;
      /* change equality constraint to ge constraint */
      for {i in NODES: _supdem_[i] < 0} balance[i].ub = infinity;
   end;

   solve with lp / solver=ns;

   num _supply_ {<i,j> in ARCS} = (if _supdem_[i] ne 0 then _supdem_[i] else .);
   num _demand_ {<i,j> in ARCS} = (if _supdem_[j] ne 0 then -_supdem_[j] else .);
   num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

   create data Solution from [_tail_ _head_]
      _cost_ _capac_ _lo_ _supply_ _demand_ _flow_=Flow _fcost_;
quit;
```

The PROC OPTMODEL statements use both single-dimensional (NODES) and multiple-dimensional (ARCS) index sets, which are populated from the corresponding data set variables in the READ DATA statements. The _SUPDEM_, _LO_, and _CAPAC_ parameters are given initial values, and the NOMISS option in the READ DATA statement tells PROC OPTMODEL to read only the nonmissing values from the input data set. The model declaration statements are a nearly direct translation from the LP formulation. The balance constraint is initially declared as an equality, but depending on the total supply or demand, the sense of this constraint is changed to "$\le$" or "$\ge$" by relaxing the constraint's lower or upper bound, respectively. Note that infinity is obtained by taking the min of the empty set. The SOLVE statement invokes the network simplex solver instead of the dual simplex solver. The Solution output data set contains the same information as provided by PROC INTPOINT.

Output 2 shows the PROC OPTMODEL output, which you can compare to the PROC OPTLP output in Output 1.

Output 2 Solving the Minimum Cost Network Flow Problem Using PROC OPTMODEL

```
                    The OPTMODEL Procedure

                       Problem Summary

Objective Sense              Minimization
Objective Function              TotalCost
Objective Type                     Linear

Number of Variables                  64
Bounded Above                         0
Bounded Below                        12
Bounded Below and Above              52
Free                                  0
Fixed                                 0

Number of Constraints                20
Linear LE (<=)                        4
Linear EQ (=)                        16
Linear GE (>=)                        0
Linear Range                          0

Constraint Coefficients             128

                    Solution Summary

Solver                    Network Simplex
Objective Function              TotalCost
Solution Status                   Optimal
Objective Value             -1281110.35
Iterations                           70
Iterations2                           6

Primal Infeasibility                  0
Dual Infeasibility                    0
Bound Infeasibility                   0
```

You can handle shortest paths, maximum flow, and generalized networks similarly, and *SAS/OR 9.3 User's Guide: Mathematical Programming Legacy Procedures* shows an example of each.

UNCAPACITATED FACILITY LOCATION

Suppose you have a set F of potential facility locations and a set C of customers to be served by these facilities. A transportation cost c_{ij} is incurred if all of customer i's demand is satisfied by facility j. Each facility $j \in F$ has a fixed opening cost f_j. The *uncapacitated facility location problem* is to determine which facilities to open and how to satisfy all customer demands at a minimum cost.

The uncapacitated facility location problem can be formulated as an MILP problem, where y_j is a binary variable that indicates whether facility j is open, and x_{ij} is the fraction of customer i's demand that is satisfied by facility j:

$$\text{minimize} \quad \sum_{i \in C} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_j y_j$$

$$\text{subject to} \quad \sum_{j \in F} x_{ij} \geq 1 \qquad \text{for } i \in C$$

$$x_{ij} \leq y_j \qquad \text{for } i \in C,\ j \in F$$

$$x_{ij} \geq 0 \qquad \text{for } i \in C,\ j \in F$$

$$y_j \in \{0, 1\} \qquad \text{for } j \in C$$

The following statements create two SAS data sets to store the input data for an example instance from Wolsey (1998):

```
/* Wolsey, Integer Programming (1998), p.181 */
%let m = 6;
data customer_data;
   do customer = 1 to &m;
      output;
   end;
run;
data cost_data;
   input fixed_cost c1-c&m;
```

```
   datalines;
 4 6  4 3 2 1 3
 8 2 10 2 0 8 2
11 1  2 4 4 6 4
 7 3  6 1 1 2 8
 5 5  1 3 4 5 1
;
run;
```

The following PROC OPTMODEL statements declare index sets and parameters, read the input data, declare the optimization model, expand the model, call the solver, print the solution, and write the solution to two data sets:

```
proc optmodel;
   /* declare index sets */
   set CUSTOMERS;
   set FACILITIES;

   /* declare parameters */
   num fixed_cost {FACILITIES};
   num unit_cost {CUSTOMERS, FACILITIES};

   /* read data sets to populate data */
   read data Customer_data into CUSTOMERS=[customer];
   read data Cost_data into FACILITIES=[_N_]
      fixed_cost
      {i in CUSTOMERS} <unit_cost[i,_N_]=col('c'||i)>;

   /* declare decision variables */
   var Assign {CUSTOMERS, FACILITIES} >= 0;
   var IsOpen {FACILITIES} binary;

   /* declare objective */
   min TotalCost =
      sum {i in CUSTOMERS, j in FACILITIES} unit_cost[i,j] * Assign[i,j]
      + sum {j in FACILITIES} fixed_cost[j] * IsOpen[j];

   /* declare constraints */
   con demand {i in CUSTOMERS}:
      sum {j in FACILITIES} Assign[i,j] >= 1;
   con link {i in CUSTOMERS, j in FACILITIES}:
      Assign[i,j] <= IsOpen[j];

   /* display expanded model */
   expand;

   /* call MILP solver (problem type automatically determined) */
   solve;

   /* print solution */
   print IsOpen Assign;

   /* write solution to data sets */
   create data Soldata1 from [j] IsOpen;
   create data Soldata2 from [i j] Assign;
quit;
```

As you can see, the PROC OPTMODEL syntax to formulate MILP problems is no more difficult than for LP. Note that the BINARY keyword is used in the IsOpen variable declaration. To declare a general integer variable, you would instead use the INTEGER keyword (with optional bounds) in the VAR statement. The “demand” constraints ensure that each customer’s demand is satisfied. The “link” constraints force IsOpen[j] = 1 if Assign[i,j] > 0. Here the MILP solver is called directly, but this example is revisited in the section “UNCAPACITATED FACILITY LOCATION USING BENDERS DECOMPOSITION” on page 10 in the discussion of advanced uses of PROC OPTMODEL.

PROGRAMMING CAPABILITIES OF THE PROC OPTMODEL MODELING LANGUAGE

The rich syntax of the PROC OPTMODEL modeling language enables you to formulate an optimization problem with ease and flexibility. The numerous expressions and statements available in PROC OPTMODEL also facilitate building customized solutions in which optimization algorithms in PROC OPTMODEL are called to solve a series of subproblems that are embedded in a larger solution context. For a complete list and description of these expressions and statements

in PROC OPTMODEL, refer to the chapter "The OPTMODEL Procedure" in *SAS/OR User's Guide: Mathematical Programming*. In the previous sections, you have seen examples that use some of these expressions and statements. This section outlines additional useful PROC OPTMODEL expressions and statements.

The following lists four groups of PROC OPTMODEL expressions:

- *index-set* expression, of which several examples were shown in the section "USING PROC OPTMODEL TO FORMULATE AND SOLVE LINEAR OPTIMIZATION PROBLEMS" on page 5
- IF-THEN/ELSE expression:

```
num _supdem_ {i in NODES} = (if i = sourcenode then 1 else if i = sinknode then -1 else 0);
```

- expressions for set operations such as subsetting, UNION, and SLICE:

```
put ({i in 1..5: i ne 3});                          /* outputs {1,2,4,5}     */
put ({1,3} union {2,3});                            /* outputs {1,3,2}       */
put (slice(<*,2,*>, {<1,2,3>, <2,4,3>, <2,2,5>})); /* outputs {<1,3>,<2,5>} */
```

- SUM and PROD expressions described in the section "USING PROC OPTMODEL TO FORMULATE AND SOLVE LINEAR OPTIMIZATION PROBLEMS" on page 5

In addition to the statements in Table 1 that describe the basic elements of an optimization problem, PROC OPTMODEL provides the following enhanced statements that describe and select multiple optimization problems:

- PROBLEM *problem* associates an objective function, decision variables, constraints, and some status information with *problem*.
- USE PROBLEM *problem* makes *problem* the current problem for the solver.

PROC OPTMODEL has the following flow control statements:

- DO loop
- DO UNTIL loop
- DO WHILE loop
- FOR loop
- IF-THEN/ELSE block
- CONTINUE, LEAVE, and STOP for loop modification

To modify an existing optimization model, you can use the following statements:

- DROP *constraint* ignores the constraint.
- RESTORE *constraint* adds back the constraint that was dropped by the DROP statement.
- FIX *variable* treats the variable as fixed in value.
- UNFIX *variable* reverses the effect of the FIX statement.

This section lists a number of commonly used expressions and statements in PROC OPTMODEL to show the versatility of the PROC OPTMODEL modeling language. The next section uses two examples to illustrate how to build customized solutions with some of these powerful expressions and statements.

ADVANCED USES OF PROC OPTMODEL

Beyond PROC OPTMODEL's use of readable and maintainable code to formulate and solve an optimization problem, the expressive programming language described in the previous section enables you to implement more powerful algorithms or customized heuristics that call an optimization solver (perhaps multiple times) as a subroutine. Some of the advanced optimization techniques implemented using the programming functionality of PROC OPTMODEL include:

- row generation (see the "Traveling Salesman Problem" example in the chapter “The Mixed Integer Linear Programming Solver” in *SAS/OR User’s Guide: Mathematical Programming*)
- column generation (see the "Multiple Subproblems" example in the chapter “The OPTMODEL Procedure” in *SAS/OR User’s Guide: Mathematical Programming*)
- feasibility pump (see Example C.2 in Huang and Hughes (2010))
- Lagrangian relaxation
- robust optimization
- multiple-objective optimization
- large-scale local search heuristic
- Dantzig-Wolfe decomposition

The following subsections show two advanced examples of linear optimization in PROC OPTMODEL:

- uncapacitated facility location problem using Benders decomposition
- blackjack as a Markov decision problem

UNCAPACITATED FACILITY LOCATION USING BENDERS DECOMPOSITION

The uncapacitated facility location problem described in the section “UNCAPACITATED FACILITY LOCATION” on page 7 has a structure that makes it particularly suitable for Benders decomposition (Benders 1962). Given a proposed set of facilities to open, the problem of optimally satisfying customer demands is trivial—just serve each customer i using the open facility j with the cheapest transportation cost c_{ij}.

The details of Benders decomposition are omitted here, but the main features are as follows:

- The algorithm iterates between solving two separate optimization problems: an MILP *master* problem and an LP *subproblem*.
- The master problem has a fixed number of variables, but the number of constraints increases.
- The optimal solution to the master problem becomes part of the objective function in the subproblem.
- The optimal dual values from the subproblem become part of the constraint matrix in the master problem.

The following PROC OPTMODEL statements solve the uncapacitated facility location problem via Benders decomposition, using the “named model” functionality provided by the PROBLEM and USE PROBLEM statements.

The first group of statements is the same as used in the direct MILP formulation described earlier, except that the PROBLEM statement is now used to name this problem ORIGINAL for later use:

```
proc optmodel;
   set CUSTOMERS;
   set FACILITIES;
   num fixed_cost {FACILITIES};
   num unit_cost {CUSTOMERS, FACILITIES};
   read data Customer_data into CUSTOMERS=[customer];
   read data Cost_data into FACILITIES=[_N_]
      fixed_cost
      {i in CUSTOMERS} <unit_cost[i,_N_]=col('c'||i)>;
   var Assign {CUSTOMERS, FACILITIES} >= 0;
   var IsOpen {FACILITIES} binary;
   min TotalCost =
      sum {i in CUSTOMERS, j in FACILITIES} unit_cost[i,j] * Assign[i,j]
      + sum {j in FACILITIES} fixed_cost[j] * IsOpen[j];
   con demand {i in CUSTOMERS}:
      sum {j in FACILITIES} Assign[i,j] >= 1;
   con link {i in CUSTOMERS, j in FACILITIES}:
      Assign[i,j] <= IsOpen[j];
   problem original include
      Assign IsOpen
      TotalCost
      demand link;
```

The next group of statements declares the LP subproblem. Although the subproblem is declared only once, the set OPEN_FACILITIES changes throughout the algorithm, based on the solution of the master problem.

```
/* declare LP subproblem */
set OPEN_FACILITIES;
var DemandDual {CUSTOMERS} >= 0;
var LinkDual {CUSTOMERS, FACILITIES} >= 0;
max SubproblemObjective =
   sum {i in CUSTOMERS} DemandDual[i]
   - sum {i in CUSTOMERS, j in OPEN_FACILITIES} LinkDual[i,j];
con dual_con {i in CUSTOMERS, j in FACILITIES}:
   DemandDual[i] - LinkDual[i,j] <= unit_cost[i,j];
problem subproblem include
   DemandDual LinkDual
   SubproblemObjective
   dual_con;
```

A nice feature of Benders decomposition is that you always have lower and upper bounds on the optimal objective value. The following statements declare numeric parameters to record this progress:

```
num infinity = min {r in {}} r;
num best_lower_bound init 0; /* lower bound on original objective TotalCost */
num best_upper_bound init infinity;
num gap;
```

Next, the MILP master problem is declared. As best_lower_bound changes, so does the lower bound of the Eta variable. Initially, the ITERATIONS index set is empty, but its size and hence also the number of constraints gradually increase.

```
/* declare MILP master problem */
num num_iters init 0;
set ITERATIONS = 1..num_iters;
num cut_constant {ITERATIONS};
num cut_coefficient {FACILITIES, ITERATIONS};
var Eta >= best_lower_bound;
min MasterObjective = Eta;
/* dynamically generated constraints */
con optimality_cut {iter in ITERATIONS}:
   Eta >= cut_constant[iter] + sum {j in FACILITIES} cut_coefficient[j,iter] * IsOpen[j];
/* necessary and sufficient conditions on IsOpen for feasibility of subproblem    */
/* (projection of original feasible region onto IsOpen space)                     */
/* can also be generated dynamically using extreme rays when subproblem unbounded */
con feasibility_cut:
   sum {j in FACILITIES} IsOpen[j] >= 1;
problem master include
   IsOpen Eta
   MasterObjective
   optimality_cut
   feasibility_cut;
```

The following DO UNTIL loop is the bulk of the algorithm:

```
num lower_bound = MasterObjective.sol;
num upper_bound = sum {j in OPEN_FACILITIES} fixed_cost[j] + SubproblemObjective.sol;
num epsilon = 1e-6;
num done init 0;
do until (done);
   put 'Solving master...';
   use problem master;
   solve;
   OPEN_FACILITIES = {j in FACILITIES: IsOpen[j].sol > 0.5};

   put 'Solving subproblem...';
   use problem subproblem;
   solve with LP / solver=ps basis=warmstart presolver=none;
   if lower_bound >= upper_bound - epsilon then do;
      put 'OPTIMAL SOLUTION FOUND';
      done = 1;
   end;
   else do;
      /* generate new optimality cut */
      num_iters = num_iters + 1;
```

```
         cut_constant[num_iters] = sum {i in CUSTOMERS} DemandDual[i].sol;
         for {j in FACILITIES}
            cut_coefficient[j,num_iters] = fixed_cost[j] - sum {i in CUSTOMERS} LinkDual[i,j].sol;
      end;

      best_lower_bound = lower_bound;
      best_upper_bound = min(best_upper_bound, upper_bound);
      gap = (if best_lower_bound > 0
            then (best_upper_bound - best_lower_bound) / best_lower_bound
            else .);
      put "BOUNDS: " best_lower_bound= best_upper_bound= gap=;
   end;
```

The first USE PROBLEM statement in this loop switches focus to the master problem. Then the MILP solver is called, and the set of open facilities in the optimal solution is recorded for use in the subproblem.

The second USE PROBLEM statement in this loop switches focus to the subproblem. Then the LP solver is called. Because changing only the objective function preserves primal feasibility, the primal simplex solver with warm start is specified to solve the subproblem; this approach can be faster than starting from scratch with the dual simplex solver. PROC OPTMODEL then automatically updates the values of lower_bound and upper_bound. If these two bounds are sufficiently close, the main loop is done. Otherwise, a new constraint is created for use in the master. Note that when num_iters is incremented, PROC OPTMODEL automatically augments the ITERATIONS index set used by the cut_constant and cut_coefficient numeric parameters and the optimality_cut constraint.

After the DO UNTIL loop terminates, the IsOpen variables are fixed to their optimal values, and the original problem is solved to determine the optimal Assign values. As in the earlier example, PROC OPTMODEL prints the optimal solution and writes this solution to two data sets.

```
   /* solve original problem with fixed optimal IsOpen to determine optimal Assign */
   put 'Solving original with IsOpen fixed...';
   use problem original;
   fix IsOpen = 0;
   for {j in OPEN_FACILITIES} fix IsOpen[j] = 1;
   solve;
   print IsOpen Assign;
   create data Soldata1 from [j] IsOpen;
   create data Soldata2 from [i j] Assign;
quit;
```

The solution is the same as the solution found in the section "UNCAPACITATED FACILITY LOCATION" on page 7. Although the statements for Benders decomposition are more complicated than the corresponding statements shown earlier to call the solver directly, some MILP problems are too large to be solved directly. Decomposition often makes such problems tractable. In some cases, you can even split the subproblem into several smaller subproblems. The main point of this example is to demonstrate that you can implement customized optimization algorithms by using the programming language provided in PROC OPTMODEL.

BLACKJACK

Because SAS Global Forum is in Las Vegas this year, an example involving the casino game blackjack seems appropriate. The LP solver is used twice in one PROC OPTMODEL call:

1. First to derive the table of conditional probabilities of the dealer attaining 17, 18, 19, 20, 21, or busting, given the dealer's up card.
2. Then, using the solution from the first step, to determine the player's optimal strategy (hit, stand, double down, or split) for each possible hand and dealer's up card.

If you are unfamiliar with the rules of blackjack, see http://en.wikipedia.org/wiki/Blackjack. Manson, Barr, and Goodnight (1975) derive the optimal strategy for four-deck blackjack. This example instead uses an "infinite-deck" approximation to simplify the calculations. That is, a "ten" (meaning a 10 or face card) is assumed to appear with probability $4/13$, and each non-ten card is assumed to appear with probability $1/13$, regardless of which cards have already been dealt.

The following SAS data sets provide the various transitions from state to state, depending on both dealer and player actions. PROC OPTMODEL reads these data sets and computes the probability of each transition.

```
/* if dealer hits with hitting_total, dealer's new total depends on next card dealt */
/* 'soft' means hand has an ace counted as 11 */
/* 'hard' means hand has no ace or that any ace must be counted as 1 to avoid busting */
data Dealer_transitions;
    input (hitting_total card1-card10) ($);
    datalines;
2       soft13  4       5       6       7       8       9       10      hard11  hard12
3       soft14  5       6       7       8       9       10      hard11  hard12  hard13
4       soft15  6       7       8       9       10      hard11  hard12  hard13  hard14
5       soft16  7       8       9       10      hard11  hard12  hard13  hard14  hard15
6       17      8       9       10      hard11  hard12  hard13  hard14  hard15  hard16
7       18      9       10      hard11  hard12  hard13  hard14  hard15  hard16  17
8       19      10      hard11  hard12  hard13  hard14  hard15  hard16  17      18
9       20      hard11  hard12  hard13  hard14  hard15  hard16  17      18      19
d10up   .       hard12  hard13  hard14  hard15  hard16  17      18      19      20
soft11  soft12  soft13  soft14  soft15  soft16  17      18      19      20      .
10      21      hard12  hard13  hard14  hard15  hard16  17      18      19      20
hard11  hard12  hard13  hard14  hard15  hard16  17      18      19      20      21
hard12  hard13  hard14  hard15  hard16  17      18      19      20      21      bust
hard13  hard14  hard15  hard16  17      18      19      20      21      bust    bust
hard14  hard15  hard16  17      18      19      20      21      bust    bust    bust
hard15  hard16  17      18      19      20      21      bust    bust    bust    bust
hard16  17      18      19      20      21      bust    bust    bust    bust    bust
soft12  soft13  soft14  soft15  soft16  17      18      19      20      21      hard12
soft13  soft14  soft15  soft16  17      18      19      20      21      hard12  hard13
soft14  soft15  soft16  17      18      19      20      21      hard12  hard13  hard14
soft15  soft16  17      18      19      20      21      hard12  hard13  hard14  hard15
soft16  17      18      19      20      21      hard12  hard13  hard14  hard15  hard16
;
run;

/* if player hits with nonpair, player's new total depends on next card dealt */
data Nonpair_transitions;
    input (nonpair card1-card10) ($);
    datalines;
2       soft13  4       5       6       7       8       9       10      hard11  hard12
3       soft14  5       6       7       8       9       10      hard11  hard12  hard13
4       soft15  6       7       8       9       10      hard11  hard12  hard13  hard14
5       soft16  7       8       9       10      hard11  hard12  hard13  hard14  hard15
6       soft17  8       9       10      hard11  hard12  hard13  hard14  hard15  hard16
7       soft18  9       10      hard11  hard12  hard13  hard14  hard15  hard16  hard17
8       soft19  10      hard11  hard12  hard13  hard14  hard15  hard16  hard17  hard18
9       soft20  hard11  hard12  hard13  hard14  hard15  hard16  hard17  hard18  hard19
10      soft21  hard12  hard13  hard14  hard15  hard16  hard17  hard18  hard19  hard20
hard11  hard12  hard13  hard14  hard15  hard16  hard17  hard18  hard19  hard20  hard21
hard12  hard13  hard14  hard15  hard16  hard17  hard18  hard19  hard20  hard21  bust
hard13  hard14  hard15  hard16  hard17  hard18  hard19  hard20  hard21  bust    bust
hard14  hard15  hard16  hard17  hard18  hard19  hard20  hard21  bust    bust    bust
hard15  hard16  hard17  hard18  hard19  hard20  hard21  bust    bust    bust    bust
hard16  hard17  hard18  hard19  hard20  hard21  bust    bust    bust    bust    bust
hard17  hard18  hard19  hard20  hard21  bust    bust    bust    bust    bust    bust
hard18  hard19  hard20  hard21  bust    bust    bust    bust    bust    bust    bust
hard19  hard20  hard21  bust    bust    bust    bust    bust    bust    bust    bust
hard20  hard21  bust    bust    bust    bust    bust    bust    bust    bust    bust
hard21  bust    bust    bust    bust    bust    bust    bust    bust    bust    bust
soft11  soft12  soft13  soft14  soft15  soft16  soft17  soft18  soft19  soft20  soft21
soft12  soft13  soft14  soft15  soft16  soft17  soft18  soft19  soft20  soft21  hard12
soft13  soft14  soft15  soft16  soft17  soft18  soft19  soft20  soft21  hard12  hard13
soft14  soft15  soft16  soft17  soft18  soft19  soft20  soft21  hard12  hard13  hard14
soft15  soft16  soft17  soft18  soft19  soft20  soft21  hard12  hard13  hard14  hard15
soft16  soft17  soft18  soft19  soft20  soft21  hard12  hard13  hard14  hard15  hard16
soft17  soft18  soft19  soft20  soft21  hard12  hard13  hard14  hard15  hard16  hard17
soft18  soft19  soft20  soft21  hard12  hard13  hard14  hard15  hard16  hard17  hard18
soft19  soft20  soft21  hard12  hard13  hard14  hard15  hard16  hard17  hard18  hard19
soft20  soft21  hard12  hard13  hard14  hard15  hard16  hard17  hard18  hard19  hard20
soft21  hard12  hard13  hard14  hard15  hard16  hard17  hard18  hard19  hard20  hard21
;
run;
```

```
/* if player does not split pair, player's total is nosplit */
data Pair_nosplit;
    input (pair nosplit) ($);
    datalines;
pair1   soft12
pair2   4
pair3   6
pair4   8
pair5   10
pair6   hard12
pair7   hard14
pair8   hard16
pair9   hard18
pair10  hard20
;
run;

/* if player splits pair, player's new total depends on next card dealt (to each hand) */
data Pair_split_transitions;
    input (pair card1-card10) ($);
    datalines;
pair1   soft12  soft13  soft14  soft15  soft16  soft17  soft18  soft19  soft20  soft21
pair2   soft13  pair2   5       6       7       8       9       10      hard11  hard12
pair3   soft14  5       pair3   7       8       9       10      hard11  hard12  hard13
pair4   soft15  6       7       pair4   9       10      hard11  hard12  hard13  hard14
pair5   soft16  7       8       9       pair5   hard11  hard12  hard13  hard14  hard15
pair6   soft17  8       9       10      hard11  pair6   hard13  hard14  hard15  hard16
pair7   soft18  9       10      hard11  hard12  hard13  pair7   hard15  hard16  hard17
pair8   soft19  10      hard11  hard12  hard13  hard14  hard15  pair8   hard17  hard18
pair9   soft20  hard11  hard12  hard13  hard14  hard15  hard16  hard17  pair9   hard19
pair10  soft21  hard12  hard13  hard14  hard15  hard16  hard17  hard18  hard19  pair10
;
run;
```

Dealer's Strategy: Markov Process

In blackjack, the dealer's strategy is fixed: hit until reaching a total of 17 or more. Given the dealer's up card, you can compute the dealer's probability of reaching each possible final total (17, 18, 19, 20, 21, or bust) by using a Markov process (Kemeny and Snell 1960). First, let D denote the dealer's state space, and let p_{ij} be the transition probability from state $i \in D$ to state $j \in D$. The following PROC OPTMODEL statements compute p_{ij}, given that the dealer does not have blackjack (because when the dealer has blackjack, the hand ends immediately and the player takes no action):

```
proc optmodel;
   set CARDS = 1..10; /* 1 = ace */
   /* probability of card c */
   num q {c in CARDS} init if c = 10 then 4/13 else 1/13;

   /*******************************************/
   /* Calculate dealer transition probabilities */
   /*******************************************/

   set UP_CARDS = {'2', '3', '4', '5', '6', '7', '8', '9', 'd10up', 'soft11'};
   set <str> HITTING_TOTALS;
   set STANDING_TOTALS = {'17', '18', '19', '20', '21', 'bust'};

   /* dealer transitions */
   str dt {HITTING_TOTALS, CARDS};
   read data Dealer_transitions into HITTING_TOTALS=[hitting_total]
      {card in CARDS} <dt[hitting_total,card]=col('card'||card)>;

   /* p[i,j] = one-step transition probability from total i to total j */
   set TOTALS = HITTING_TOTALS union STANDING_TOTALS;
   num p {TOTALS, TOTALS} init 0;
   for {i in HITTING_TOTALS, j in TOTALS} p[i,j] = sum {c in CARDS: dt[i,c] = j} q[c];
   /* condition on dealer not having blackjack */
   for {j in TOTALS} do;
      p['d10up',j] = p['d10up',j] / (1 - q[1]);
      p['soft11',j] = p['soft11',j] / (1 - q[10]);
```

```
end;
/* absorbing states */
for {i in STANDING_TOTALS} p[i,i] = 1;
```

Now let π_{ij} be the conditional probability of the dealer reaching state $j \in D$, starting in state $i \in D$, given that the dealer does not have blackjack. Then

$$\pi_{ij} = \sum_k p_{ik}\pi_{kj} \quad \text{for } i \in D,\ j \in D \tag{1}$$

$$\pi_{ii} = 1 \quad \text{for } i \in \{17, 18, 19, 20, 21, \text{bust}\} \tag{2}$$

$$\sum_j \pi_{ij} = 1 \quad \text{for } i \in D \tag{3}$$

After calculating the transition probabilities p_{ij}, you can define a dummy objective and use the LP solver in PROC OPTMODEL to solve the linear system (1)–(3) to obtain the absorption probabilities π_{ij} as follows:

```
/*****************************************/
/* Compute dealer absorption probabilities */
/*****************************************/

/* Pi[i,j] = probability of attaining total j, starting with total i,
   conditional on no dealer blackjack */
var Pi {TOTALS, TOTALS} >= 0 <= 1;

con recurrence {i in HITTING_TOTALS, j in TOTALS}:
   Pi[i,j] = sum {k in TOTALS} p[i,k] * Pi[k,j];

con absorbing {i in STANDING_TOTALS}:
   Pi[i,i] = 1;

con sum_to_one {i in TOTALS}:
   sum {j in TOTALS} Pi[i,j] = 1;

/* dummy objective */
min Objective1 = 0;

put 'Computing dealer absorption probabilities...';
solve;

print 'Dealer absorption probabilities (conditional on no dealer blackjack)';
print {i in UP_CARDS, j in STANDING_TOTALS} Pi[i,j];
```

Note that the constraint declarations match the mathematical formulation very closely. Output 3 shows the resulting values of π_{ij}. As intuition might suggest, a 6 is the up card most likely to result in the dealer busting.

Output 3 Dealer Absorption Probabilities (Conditional on No Dealer Blackjack)

The OPTMODEL Procedure

Dealer absorption probabilities (conditional on no dealer blackjack)

	Pi					
	17	18	19	20	21	bust
2	0.139809	0.134907	0.129655	0.124026	0.117993	0.353608
3	0.135034	0.130482	0.125581	0.120329	0.114700	0.373875
4	0.130490	0.125938	0.121386	0.116485	0.111233	0.394468
5	0.122251	0.122251	0.117700	0.113148	0.108246	0.416404
6	0.165438	0.106267	0.106267	0.101715	0.097163	0.423150
7	0.368566	0.137797	0.078625	0.078625	0.074074	0.262312
8	0.128567	0.359336	0.128567	0.069395	0.069395	0.244741
9	0.119995	0.119995	0.350765	0.119995	0.060824	0.228425
d10up	0.120710	0.120710	0.120710	0.370710	0.037376	0.229785
soft11	0.188917	0.188917	0.188917	0.188917	0.077806	0.166525

Player's Strategy: Markov Decision Problem

Unlike the dealer, the player has several choices about what actions to take, and the transition probabilities depend on these actions. You can model this situation as a Markov decision problem (Puterman 1994). Let S denote the state

space, and let $p_{st}(a)$ be the transition probability from state $s \in S$ to state $t \in S$ if the player takes action $a \in A_s$ (available actions depend on the state), given that the dealer does not have blackjack. Define the value function $V(s)$ to be the player's expected profit under optimal play, starting in state s. Then V satisfies Bellman's equation:

$$V(s) = \max_{a \in A_s} \sum_t p_{st}(a) V(t) \quad \text{for } s \in S \tag{4}$$

To solve (4), you can use the following LP formulation:

$$\text{minimize} \quad \sum_s V(s) \tag{5}$$

$$\text{subject to} \quad V(s) \geq \sum_t p_{st}(a) V(t) \quad \text{for } s \in S,\ a \in A_s \tag{6}$$

$$V(s) \text{ free} \quad \text{for } s \in S \tag{7}$$

The optimal dual variables then indicate which action to take in each state.

For blackjack, the states are (i, u, d), where i is the player's hand, u is the dealer's up card, and $d = 1$ if the player can double ($d = 0$ otherwise). Here, i indicates the total number of points in the player's hand, whether the hand is "hard" or "soft" (contains an ace counted as 11), and whether the hand is a pair of cards of the same rank. You can use the LP solver in PROC OPTMODEL to solve (5)–(7). Note that the values of π_{ij} are fixed and the constraints that correspond to (1)–(3) are dropped. For readability and maintainability, the constraints are grouped by action (hit, stand, double, split). You can easily accommodate various differences in casino rules (dealer hits on soft 17, player can resplit aces, double after split, and so on) by modifying the constraints.

```
/**********************************/
/* Compute optimal player strategy */
/**********************************/

fix Pi;
drop recurrence absorbing sum_to_one;

/* player transitions from nonpairs, depending on next card dealt */
set <str> NONPAIRS;
str pt_nonpairs {NONPAIRS, CARDS};
read data Nonpair_transitions into NONPAIRS=[nonpair]
   {card in CARDS} <pt_nonpairs[nonpair,card]=col('card'||card)>;

/* player transitions from unsplit pairs */
set <str> PAIRS;
str pt_nosplit {PAIRS};
read data Pair_nosplit into PAIRS=[pair] pt_nosplit=nosplit;

/* player transitions from split pairs, depending on next card dealt */
str pt_split {PAIRS, CARDS};
read data Pair_split_transitions into [pair]
   {card in CARDS} <pt_split[pair,card]=col('card'||card)>;

/* number of points in player's or dealer's final hand */
num points {t in NONPAIRS union STANDING_TOTALS union {'soft12'}} =
   if t = 'bust' then 0 else input(compress(t,'hardsoft'),best.);

/* exclude intermediate states where no decision is made */
set NONPAIRS2 = NONPAIRS diff {'2', '3', 'soft11'};
set PLAYER_TOTALS = PAIRS union NONPAIRS2 union {'bust'};

/* value function: V[i,u,d] = expected profit under optimal play,
   i = player's total, u = dealer's up card, d = 1 if player can double */
var V {PLAYER_TOTALS, UP_CARDS, 0..1};
/* Bellman's equation: V[s] = max {a in A[s]} sum {t} prob[s,t,a] * V[t] for all s */

/* dummy objective (for Markov decision problem, all coefficients must be positive) */
min Objective2 = sum {i in PLAYER_TOTALS, u in UP_CARDS, d in 0..1} V[i,u,d];

/* if hit, then take at least one more card */
/* cannot double after hit */
con hit {i in NONPAIRS2, u in UP_CARDS, d in 0..1}:
   V[i,u,d] >= sum {c in CARDS} q[c] * V[pt_nonpairs[i,c],u,0];

/* if stand, then expected profit is prob(win) - prob(lose) */
```

```
con stand {i in NONPAIRS2, u in UP_CARDS, d in 0..1}:
   V[i,u,d] >=
      sum {j in STANDING_TOTALS: points[j] < points[i]} Pi[u,j]
      - sum {j in STANDING_TOTALS: points[j] > points[i]} Pi[u,j];

/* if double down, then expected profit is 2 * (prob(win) - prob(lose)) */
/* can double down only if d = 1 */
con double {i in NONPAIRS2, u in UP_CARDS}:
   V[i,u,1] >=
      2 * sum {c in CARDS} q[c] * (
      sum {j in STANDING_TOTALS: points[j] < points[pt_nonpairs[i,c]]} Pi[u,j]
      - sum {j in STANDING_TOTALS:
         points[j] > points[pt_nonpairs[i,c]] or pt_nonpairs[i,c] = 'bust'} Pi[u,j]
      );

/* if split, then double bet and take at least one more card on each hand,
   exactly one more card for split aces */
/* allow infinite resplitting for all pairs, except no resplitting of aces */
con split {i in PAIRS, u in UP_CARDS, d in 0..1}:
   V[i,u,d] >=
      if i ne 'pair1'
      then 2 * sum {c in CARDS} q[c] * V[pt_split[i,c],u,1]
      else 2 * sum {c in CARDS} q[c] * (
         sum {j in STANDING_TOTALS: points[j] < points[pt_split[i,c]]} Pi[u,j]
         - sum {j in STANDING_TOTALS: points[j] > points[pt_split[i,c]]} Pi[u,j]
      );

/* if no split, then advance to nonpair state */
con nosplit {i in PAIRS, u in UP_CARDS, d in 0..1}:
   V[i,u,d] >= V[pt_nosplit[i],u,d];

/* player bust always loses, even if dealer also busts */
con bust {u in UP_CARDS, d in 0..1}:
   V['bust',u,d] = -1;

put 'Computing optimal strategy...';
solve;

print 'Player optimal value function V[i,u,1]';
print {i in PLAYER_TOTALS, u in UP_CARDS} (round(V[i,u,1].sol,1e-4));
create data V1 from [player_total]=PLAYER_TOTALS
   {u in UP_CARDS} <col('card'||u)=V[player_total,u,1]>;
```

Output 4 shows the optimal solution returned by the LP solver, with positive (or negative) values indicating expected winning (or losing) hands for the player.

Output 4 Player Optimal Value Function $V(i, u, 1)$

Player optimal value function V[i,u,1]

	2	3	4	5	6	7	8	9	d10up	soft11
10	0.3589	0.4093	0.4609	0.5125	0.5756	0.3924	0.2866	0.1443	0.0253	0.0814
4	-0.1149	-0.0826	-0.0494	-0.0124	0.0111	-0.0883	-0.1593	-0.2407	-0.2892	-0.2531
5	-0.1282	-0.0953	-0.0615	-0.0240	-0.0012	-0.1194	-0.1881	-0.2666	-0.3134	-0.2786
6	-0.1408	-0.1073	-0.0729	-0.0349	-0.0130	-0.1519	-0.2172	-0.2926	-0.3377	-0.3041
7	-0.1092	-0.0766	-0.0430	-0.0073	0.0292	-0.0688	-0.2106	-0.2854	-0.3191	-0.3101
8	-0.0218	0.0080	0.0388	0.0708	0.1150	0.0822	-0.0599	-0.2102	-0.2494	-0.1970
9	0.0744	0.1208	0.1819	0.2431	0.3171	0.1719	0.0984	-0.0522	-0.1530	-0.0657
bust	-1.0000	-1.0000	-1.0000	-1.0000	-1.0000	-1.0000	-1.0000	-1.0000	-1.0000	-1.0000
hard11	0.4706	0.5178	0.5660	0.6147	0.6674	0.4629	0.3507	0.2278	0.1797	0.1430
hard12	-0.2534	-0.2337	-0.2111	-0.1672	-0.1537	-0.2128	-0.2716	-0.3400	-0.3810	-0.3505
hard13	-0.2928	-0.2523	-0.2111	-0.1672	-0.1537	-0.2691	-0.3236	-0.3872	-0.4253	-0.3969
hard14	-0.2928	-0.2523	-0.2111	-0.1672	-0.1537	-0.3213	-0.3719	-0.4309	-0.4663	-0.4400
hard15	-0.2928	-0.2523	-0.2111	-0.1672	-0.1537	-0.3698	-0.4168	-0.4716	-0.5044	-0.4800
hard16	-0.2928	-0.2523	-0.2111	-0.1672	-0.1537	-0.4148	-0.4584	-0.5093	-0.5398	-0.5171
hard17	-0.1530	-0.1172	-0.0806	-0.0449	0.0117	-0.1068	-0.3820	-0.4232	-0.4197	-0.4780
hard18	0.1217	0.1483	0.1759	0.1996	0.2834	0.3996	0.1060	-0.1832	-0.1783	-0.1002
hard19	0.3863	0.4044	0.4232	0.4395	0.4960	0.6160	0.5939	0.2876	0.0631	0.2776
hard20	0.6400	0.6503	0.6610	0.6704	0.7040	0.7732	0.7918	0.7584	0.5545	0.6555
hard21	0.8820	0.8853	0.8888	0.8918	0.9028	0.9259	0.9306	0.9392	0.9626	0.9222
pair1	0.4706	0.5178	0.5660	0.6147	0.6674	0.4629	0.3507	0.2278	0.1797	0.1091
pair10	0.6400	0.6503	0.6610	0.6704	0.7040	0.7732	0.7918	0.7584	0.5545	0.6555
pair2	-0.0842	-0.0153	0.0597	0.1526	0.2282	0.0073	-0.1593	-0.2407	-0.2892	-0.2531
pair3	-0.1377	-0.0560	0.0305	0.1272	0.2026	-0.0525	-0.2172	-0.2926	-0.3377	-0.3041
pair4	-0.0218	0.0080	0.0388	0.0820	0.1516	0.0822	-0.0599	-0.2102	-0.2494	-0.1970
pair5	0.3589	0.4093	0.4609	0.5125	0.5756	0.3924	0.2866	0.1443	0.0253	0.0814
pair6	-0.2123	-0.1190	-0.0202	0.0824	0.1555	-0.2128	-0.2716	-0.3400	-0.3810	-0.3505
pair7	-0.1305	-0.0425	0.0508	0.1485	0.2498	-0.0485	-0.3719	-0.4309	-0.4663	-0.4400
pair8	0.0760	0.1485	0.2234	0.3002	0.4127	0.3254	-0.0202	-0.3865	-0.4803	-0.3717
pair9	0.1961	0.2592	0.3243	0.3931	0.4725	0.3996	0.2352	-0.0774	-0.1783	-0.1002
soft12	0.0818	0.1035	0.1266	0.1565	0.1860	0.1655	0.0951	0.0001	-0.0700	-0.0205
soft13	0.0466	0.0741	0.1025	0.1334	0.1797	0.1224	0.0541	-0.0377	-0.1049	-0.0573
soft14	0.0224	0.0508	0.0801	0.1260	0.1797	0.0795	0.0133	-0.0752	-0.1395	-0.0939
soft15	-0.0001	0.0292	0.0593	0.1260	0.1797	0.0370	-0.0271	-0.1122	-0.1737	-0.1300
soft16	-0.0210	0.0091	0.0584	0.1260	0.1797	-0.0049	-0.0668	-0.1486	-0.2074	-0.1656
soft17	-0.0005	0.0551	0.1187	0.1824	0.2561	0.0538	-0.0729	-0.1498	-0.1969	-0.1796
soft18	0.1217	0.1776	0.2370	0.2952	0.3815	0.3996	0.1060	-0.1007	-0.1438	-0.0929
soft19	0.3863	0.4044	0.4232	0.4395	0.4960	0.6160	0.5939	0.2876	0.0631	0.2776
soft20	0.6400	0.6503	0.6610	0.6704	0.7040	0.7732	0.7918	0.7584	0.5545	0.6555
soft21	0.8820	0.8853	0.8888	0.8918	0.9028	0.9259	0.9306	0.9392	0.9626	0.9222

Four starting states (for example, hard16 versus a dealer ten) have $V(s) < -0.5$, so you would do better to "surrender" (give up half your bet instead of continuing to play the hand) in casinos that allow that option.

By using the .dual constraint suffix, you can access the optimal dual solution and hence the optimal action to take in each state.

```
/* dual variables indicate optimal actions */
print 'Optimal strategy for pairs';
print '(Y = split, else do not split)';
print {i in PAIRS, u in UP_CARDS} (
   if split[i,u,1].dual > 0 then 'Y'
   else ''
);
print 'Optimal strategy if can double down';
print '(D = double down, H = hit, else stand)';
print {i in NONPAIRS2, u in UP_CARDS} (
   if double[i,u].dual > 0 then 'D'
   else if hit[i,u,1].dual > 0 then 'H'
   else ''
);
print 'Optimal strategy if cannot double down';
print '(H = hit, else stand)';
print {i in NONPAIRS2, u in UP_CARDS} (
   if hit[i,u,0].dual > 0 then 'H'
   else ''
);
```

Output 5, Output 6, and Output 7 show the optimal strategy.

Output 5 Optimal Strategy for Pairs

Optimal strategy for pairs

(Y = split, else do not split)

	2	3	4	5	6	7	8	9	d10up	soft11
pair1	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
pair10										
pair2	Y	Y	Y	Y	Y	Y				
pair3	Y	Y	Y	Y	Y	Y				
pair4				Y	Y					
pair5										
pair6	Y	Y	Y	Y	Y					
pair7	Y	Y	Y	Y	Y	Y				
pair8	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
pair9	Y	Y	Y	Y	Y		Y	Y		

Output 6 Optimal Strategy If You Can Double Down

Optimal strategy if can double down

(D = double down, H = hit, else stand)

	2	3	4	5	6	7	8	9	d10up	soft11
10	D	D	D	D	D	D	D	D	H	H
4	H	H	H	H	H	H	H	H	H	H
5	H	H	H	H	H	H	H	H	H	H
6	H	H	H	H	H	H	H	H	H	H
7	H	H	H	H	H	H	H	H	H	H
8	H	H	H	H	H	H	H	H	H	H
9	H	D	D	D	D	H	H	H	H	H
hard11	D	D	D	D	D	D	D	D	D	H
hard12	H	H				H	H	H	H	H
hard13						H	H	H	H	H
hard14						H	H	H	H	H
hard15						H	H	H	H	H
hard16						H	H	H	H	H
hard17										
hard18										
hard19										
hard20										
hard21										
soft12	H	H	H	H	H	H	H	H	H	H
soft13	H	H	H	H	D	H	H	H	H	H
soft14	H	H	H	D	D	H	H	H	H	H
soft15	H	H	H	D	D	H	H	H	H	H
soft16	H	H	D	D	D	H	H	H	H	H
soft17	H	D	D	D	D	H	H	H	H	H
soft18		D	D	D	D			H	H	H
soft19										
soft20										
soft21										

Output 7 Optimal Strategy If You Cannot Double Down

Optimal strategy if cannot double down

(H = hit, else stand)

	2	3	4	5	6	7	8	9	d10up	soft11
10	H	H	H	H	H	H	H	H	H	H
4	H	H	H	H	H	H	H	H	H	H
5	H	H	H	H	H	H	H	H	H	H
6	H	H	H	H	H	H	H	H	H	H
7	H	H	H	H	H	H	H	H	H	H
8	H	H	H	H	H	H	H	H	H	H
9	H	H	H	H	H	H	H	H	H	H
hard11	H	H	H	H	H	H	H	H	H	H
hard12	H	H				H	H	H	H	H
hard13						H	H	H	H	H
hard14						H	H	H	H	H
hard15						H	H	H	H	H
hard16						H	H	H	H	H
hard17										
hard18										
hard19										
hard20										
hard21										
soft12	H	H	H	H	H	H	H	H	H	H
soft13	H	H	H	H	H	H	H	H	H	H
soft14	H	H	H	H	H	H	H	H	H	H
soft15	H	H	H	H	H	H	H	H	H	H
soft16	H	H	H	H	H	H	H	H	H	H
soft17	H	H	H	H	H	H	H	H	H	H
soft18								H	H	H
soft19										
soft20										
soft21										

By weighting the $V(s)$ values according to the probability of starting in state s and incorporating the possibility of a dealer blackjack or a player blackjack, you can compute the total expected profit under the optimal strategy:

```
/************************************************/
/* Compute expected profit for optimal strategy */
/************************************************/

set NONPAIRS3 = NONPAIRS2 diff {'4', 'soft12'};
set STARTING_TOTALS = PAIRS union NONPAIRS3;

/* probability of being dealt each starting total */
num init_prob {STARTING_TOTALS};
for {i in PAIRS} init_prob[i] = q[input(compress(i,'pair'),best.)]^2;
for {i in NONPAIRS3} do;
   if index(i,'soft') > 0 then init_prob[i] =
      (if i = 'soft21'
      then 2 * q[1] * q[10] else 2 * q[1] * q[input(compress(i,'soft'),best.)-11]);
   else init_prob[i] =
      sum {k in 2..10: points[i]-k in 2..10 diff {k}} q[k] * q[points[i]-k];
end;

print 'Probability of being dealt each starting total';
print init_prob;

num card {u in UP_CARDS} init mod(input(compress(u,'dupsoft'),best.)-1,10)+1;
/* conditional probability of dealer's up card c, given dealer does not have blackjack */
num q2 {c in CARDS} init q[c];
/* if ace, then no ten */
q2[1] = q[1] * (1 - q[10]);
/* if ten, then no ace */
q2[10] = q[10] * (1 - q[1]);
for {c in CARDS} q2[c] = q2[c] / (1 - init_prob['soft21']);

print 'Expected profit for optimal strategy';
num expected_profit =
   /* dealer blackjack but not player blackjack */
   init_prob['soft21'] * (1 - init_prob['soft21']) * (-1)
```

```
      /* no dealer blackjack */
      + (1 - init_prob['soft21']) * (
      sum {i in STARTING_TOTALS diff {'soft21'}, u in UP_CARDS}
         init_prob[i] * q2[card[u]] * V[i,u,1].sol
      + init_prob['soft21'] * 1.5
   );
   print expected_profit;
quit;
```

This total expected profit turns out to be approximately -0.005, an expected loss of half a cent per dollar bet.

For this application, the LP presolver solves the problem instantly, but you can use the same ideas and similar statements to solve larger Markov decision problems that require the LP solver to do more work.

CONCLUSION

This paper describes the linear optimization solvers available in PROC OPTMODEL and gives several examples of their use. Compared with the legacy SAS/OR linear optimization procedures, PROC OPTMODEL provides much more readable and maintainable code with direct access to more powerful solvers. For new projects or projects that require more manageable code or faster run times, PROC OPTMODEL is recommended for linear optimization. Legacy procedure users can also try out the new solvers on their problems by using the MPSOUT= option in combination with either the OPTLP or the OPTMILP procedure.

REFERENCES

Benders, J. F. (1962), "Partitioning Procedures for Solving Mixed-Variables Programming Problems," *Numerische Mathematik*, 4, 238–252.

Huang, T. and Hughes, E. (2010), "Nonlinear Optimization in SAS/OR® Software: Migrating from PROC NLP to PROC OPTMODEL," in *Proceedings of the SAS Global Forum 2010 Conference*, Cary, NC: SAS Institute Inc.

Kemeny, J. G. and Snell, J. L. (1960), *Finite Markov Chains*, The University Series in Undergraduate Mathematics, Princeton, NJ: D. Van Nostrand.

Manson, A. R., Barr, A. J., and Goodnight, J. H. (1975), "Optimum Zero-Memory Strategy and Exact Probabilities for 4-Deck Blackjack," *The American Statistician*, 29(2), pp. 84–88.

Puterman, M. L. (1994), *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, Wiley Series in Probability and Statistics, New York: Wiley.

Wolsey, L. A. (1998), *Integer Programming*, New York: Wiley-Interscience.

CONTACT INFORMATION

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

Rob Pratt
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
E-mail: Rob.Pratt@sas.com

Ed Hughes
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
E-mail: Ed.Hughes@sas.com