FICO™

# Embedding Optimization Algorithms

## FICO™ Xpress Optimization Suite Whitepaper

Last update 22 January, 2007

# Embedding Optimization Algorithms


## Susanne Heipcke

Xpress Team, FICO, Leam House, Leamington Spa CV32 5YN, UK
http://www.fico.com/xpress


November 2002, last rev. January 2007


### Abstract

This paper shows how to use the modeling and solving environment Mosel to implementing complex algorithms directly in the modeling language. To demonstrate the flexibility of the Mosel language and its open interface to external software we discuss several practical examples of interaction between models and external programs.
"Traditional embedding" is used to solve a transport problem with a set of demand scenarios, followed by an analysis of the results. The "Office cleaning" problem is solved by a branch-and-cut algorithm, implemented entirely in the modeling language (including the definition of the callback function for the solver). And finally, a cutting stock problem is solved by generating columns (= cutting patterns), also implemented in the modeling language. In this example, we input data in memory, which means that the model accesses information from the calling program during its execution, or, looking at it from a different angle, the calling program modifies the data held in the model.


**Keywords:** Mathematical Programming, Mathematical Programming algorithms, Programming languages

## Contents


1 Introduction . . . 2
2 Traditional embedding: solving multiple scenarios . . . 2
 2.1 Transport problem . . . 3
 2.2 Implementation . . . 3
  2.2.1 Mosel model . . . 3
  2.2.2 C program . . . 5
3 Office cleaning: callbacks for branch-and-cut . . . 5
 3.1 Office cleaning problem . . . 5
 3.2 Implementation . . . 7
 3.3 Results . . . 8
4 Column generation for cutting-stock . . . 9
 4.1 Cutting-stock problem . . . 9
 4.2 Implementation . . . 10
  4.2.1 Mosel model . . . 10
  4.2.2 C program . . . 13
 4.3 Results for bin packing benchmark problems . . . 13
 4.4 Simplified data exchange in memory using I/O drivers . . . 16
5 Conclusion . . . 17
Appendix: code listings . . . 19
 Office cleaning problem . . . 19
 Cutting-stock Problem . . . 22



©2009 Fair Isaac Corporation. All rights reserved.

# 1 Introduction

A typical OR project involves the analysis of the problem at hand, the development of a model, followed by its solution and verification. For its deployment the model then needs to be embedded into the company information system. Different commercial modeling software packages provide solutions to this issue via external additions to their algebraic modeling language: this may be scripting languages (*e.g.* AMPL [5] or OPL (1998) [11] with OPL-script) or programming language interfaces (*e.g.* EMOSL [10] for mp-model). It is also possible to use model building tools like LP-Toolkit or Xpress-BCL with which a model is formulated and solved directly in a programming language environment (such as C, C++, or Java) – at the expense of the high level language support provided by a modeling language.

In this paper we show new possibilities for embedding models and interacting with them from external programs using Xpress-Mosel [2], [3].

Section 2 gives an example of what may be called "traditional embedding": this type of embedding and interaction is the functionality provided by the systems cited above. The following two sections show new ways of interacting with a model that have become possible through Mosel's flexible open interface to external software. The primary subject of Section 3 is the communication between the model and a solver during the solution process. Section 4 shows how the information held in or produced by a model during its execution can be accessed and modified by an external program.

# 2 Traditional embedding: solving multiple scenarios

Many industrial applications that include mathematical models are based on (re-)running a parameterized model that reads data from different files. A typical calling sequence for an application with an embedded mathematical model (for instance written with mp-model) is shown in Figure 1: the application first calls the model to generate a matrix, it then starts the solution process and finally retrieves the results, again through the model.

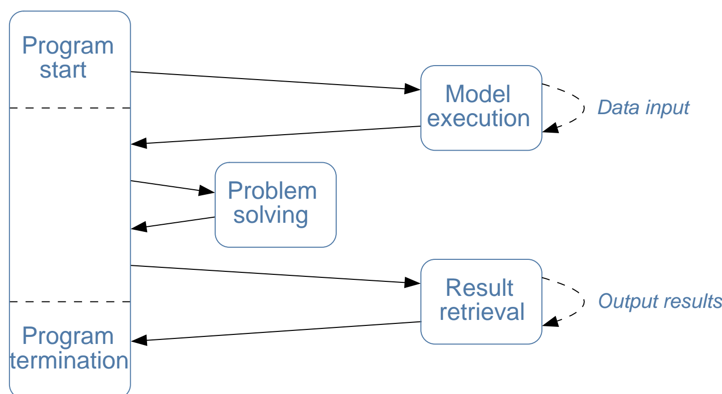


**Figure 1**: "Traditional embedding" of a model for deployment

Embedding a Mosel program into an application follows a slightly different scheme (Figure 2): the optimization process is started from the model, the problem matrix is loaded in memory into the solver (leading to a significant speed-up for large problems), and the results can be retrieved in the same model execution.


©2009 Fair Isaac Corporation. All rights reserved.

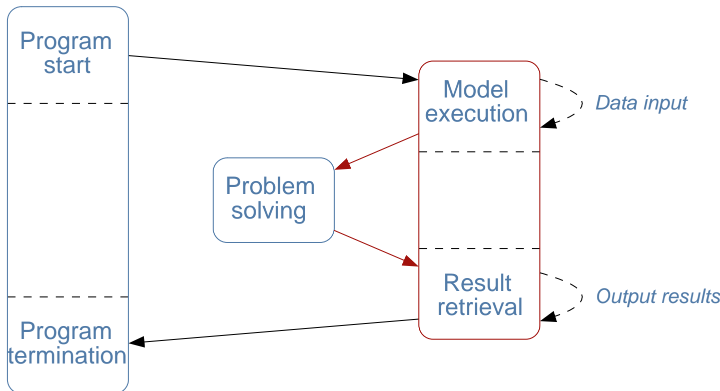


**Figure 2**: Embedding a Mosel program

## 2.1 Transport problem

As an example, let us consider a simple transport problem: a set of suppliers (*Suppliers*) needs to deliver to a set of customers *Customers*. For all feasible supplier-customer pairs $(s, c)$ the cost per unit of product $COST_{sc}$ is given. We also know the production capacity $CAP_s$ of every supplier $s$. Let $DEM_c$ be the demand by customer $c$. We wish to determine the quantities of product $x_{sc}$ to transport from supplier $s$ to customer $c$ that minimize the total cost:

$$\text{minimize} \sum_{s \in Suppliers} \sum_{\substack{c \in Customers \\ \exists COST_{sc}}} COST_{sc} \cdot x_{sc}$$

$$\forall s \in Suppliers: \sum_{c \in Customers} x_{sc} \leq CAP_s$$

$$\forall c \in Customers: \sum_{s \in Suppliers} x_{sc} \geq DEM_c$$

## 2.2 Implementation

### 2.2.1 Mosel model

The translation of the mathematical model into a Mosel program (Figure 3) is fairly straightforward, but some implementation details may necessitate an explanation:

**`uses "mmxprs"`** Specify the Xpress-Optimizer to solve the optimization problem.

**`parameters`** If we wish to use a different demand data set, a new value of the parameter `DEMFILE` can be specified when executing the model.

**`declarations`** In this block we declare the data and decision variable arrays. Note that their sizes are not given, the arrays are therefore *dynamic*.

**`initializations`** The problem data are read from two different files; the initialization of the arrays also defines their index sets.

**`create`** We explicitly create only those decision variables that are required. Any sums over this array of variables will only sum up the defined entries. However, in the definition of the objective function `MinCost` we repeat the test of the existence of $COST_{sc}$ to obtain a more efficient enumeration.


©2009 Fair Isaac Corporation. All rights reserved.

```
model "Scenario"
  uses "mmxprs"                                  !  Use Xpress-Optimizer

  parameters
    DEMFILE="Data/dem1.dat"
  end-parameters

  declarations
    Suppliers:  set of string                    !  Set of suppliers
    Customers:  set of string                    !  Set of customers
    COST: array(Suppliers,Customers) of real     !  Cost per supplier-customer pair
    CAP: array(Suppliers) of real                !  Production capacity
    DEM: array(Customers) of real                !  Demand by customers
    x:  array(Suppliers,Customers) of mpvar      !  Transported quantities
  end-declarations

  initializations from "Data/supply.dat"
    COST CAP
  end-initializations
  initializations from DEMFILE
    DEM
  end-initializations

  forall(s in Suppliers, c in Customers | exists(COST(s,c))) create(x(s,c))

  !  Objective:  minimize total cost
  MinCost:= sum(s in Suppliers, c in Customers | exists(COST(s,c))) COST(s,c)*x(s,c)

  !  Limits on production capacities
  forall(s in Suppliers) sum(c in Customers) x(s,c) <= CAP(s)

  !  Satisfy customer demands
  forall(c in Customers) sum(s in Suppliers) x(s,c) >= DEM(c)

  !  Solve the problem
  minimize(MinCost)

  !  Solution printing
  if(getprobstat=XPRS_OPT) then
  writeln("Total cost:  ", getobjval)
  forall(s in Suppliers) writeln(s, ":  ", getsol(sum(c in Customers) x(s,c)))
  end-if
end-model
```

**Figure 3**: Mosel model: transportation problem


©2009 Fair Isaac Corporation. All rights reserved.

### 2.2.2 C program

In this example, the customer demands $DEM_c$ are given in the form of five scenarios with weights $WEIGHT_i$ corresponding to the probability of scenario $i$. We want to run the model with the different data sets and analyse the solution: calculate the cost forecast and the production capacities used (weighted averages of scenario results for feasible problem instances). This is done by the following C program (Figure 4).

# 3 Office cleaning: callbacks for branch-and-cut

With Mosel, it is possible to interact with the model during its execution. In particular, a solution algorithm (or any other external program) may call subroutines that are defined in a Mosel model. During a MIP branch-and-bound search, for instance, one may wish to save the integer solutions found, direct the construction of the search tree, or add cuts to the problem. In Mosel, this is possible via the definition of *callback functions*, that is, subroutines defined in a model that are called from an external program such as a solver (Figure 5).

## 3.1 Office cleaning problem

As an example, we discuss a branch-and-cut algorithm for the "Office cleaning problem" ([7]): a large company is planning to outsource the cleaning of its offices at the least cost. The $NSITES$ office sites of the company are grouped into areas (set $AREAS = \{1, \ldots, NAREAS\}$). Several professional cleaning companies (set $CONTR = \{1, \ldots, NCONTR\}$) have submitted bids for the different sites, a cost of 0 in the data meaning that a contractor is not bidding for a site.

To avoid being dependent on a single contractor, adjacent areas ($a$,$b$ adjacent $\Leftrightarrow ADJACENT_{ab} = 1$) have to be allocated to different contractors. Every site $s$ in $SITES = \{1, \ldots, NSITES\}$ is to be allocated to a single contractor, but there may be between $LOWCON_a$ and $UPPCON_a$ contractors per area $a$.

For the mathematical formulation of the problem we introduce two sets of decision variables:
$clean_{cs}$ indicates whether contractor $c$ is cleaning site $s$
$alloc_{ca}$ indicates whether contractor $c$ is allocated any site in area $a$

The objective to minimize the total cost of all contracts is as follows (where $PRICE_{sc}$ is the price per site and contractor):

$$\text{minimize} \sum_{c \in CONTR} \sum_{s \in SITES} PRICE_{sc} \cdot clean_{cs}$$

We also have the following sets of constraints:

- Each site must be cleaned by exactly one contractor.

$$\forall s \in SITES : \sum_{c \in CONTR} clean_{cs} = 1$$

- Adjacent areas must not be allocated to the same contractor.

$$\forall c \in CONTR, a, b \in AREAS, a > b \text{ and } ADJACENT_{ab} = 1 : \\ alloc_{ca} + alloc_{cb} \leq 1$$

- The lower and upper limits on the number of contractors per area must be respected.

$$\forall a \in AREAS : \sum_{c \in CONTR} alloc_{ca} \geq LOWCON_a$$


©2009 Fair Isaac Corporation. All rights reserved.

```
#include <stdio.h>
#include <string.h>
#include "xprm_mc.h"

#define NSUP 6

double obj, capuse[NSUP], CAP[NSUP];
char SNAME[NSUP][64];

int execscenar(const char *FILENAME, double WEIGHT) {
  XPRMmodel mod;
  XPRMalltypes rval;
  XPRMarray darr, varr;
  XPRMset set[1];
  XPRMmpvar x;
  int indices[2], result;
  char params[128];

  sprintf(params, "DEMFILE=Data/%s", FILENAME);

                                       /* Execute the model file 'scenar.mos'*/
  if(XPRMexecmod(NULL, "scenar.mos", params, &result, &mod))
  { printf("Problem %s not executed.\n", FILENAME); return 0; }

                                       /* Test whether a solution is found */
  if((XPRMgetprobstat(mod)&XPRM_PBRES)!=XPRM_PBOPT)
  { printf("Demand scenario %s is infeasible.\n", FILENAME); return 0; }

  obj += WEIGHT*XPRMgetobjval(mod);         /* Get the objective function value */

/* At first execution:  get 'CAP' data and 'Suppliers' index set for solution printout */
  if(!strcmp(FILENAME, "dem1.dat")) {
    XPRMfindident(mod, "CAP", &rval);       /* Get the model object 'CAP' */
    darr = rval.array;
    XPRMgetarrsets(darr, set);              /* Get the index set of 'CAP' */
    XPRMgetfirstarrentry(darr, indices);    /* Get the first index tuple */
    do {                                    /* (one-dimensional dense array) */
      strcpy(SNAME[indices[0]-1],
        XPRMgetelsetval(set[0], indices[0], &rval)->string);
      XPRMgetarrval(darr, indices, &(CAP[indices[0]-1]));
    } while(!XPRMgetnextarrentry(darr, indices));
  }                                         /* Get the next index tuple */

/* At every execution:  get the solution for variables 'x' */
  XPRMfindident(mod, "x", &rval);           /* Get the model object 'x' */
  varr = rval.array;
  XPRMgetfirstarrtruentry(varr, indices);   /* Get the first entry of array varr */
  do {                                      /* (two-dimensional sparse array) */
    XPRMgetarrval(varr, indices, &x);       /* Get the corresponding variable */
    capuse[indices[0]-1] += WEIGHT*XPRMgetvsol(mod, x);
  } while(!XPRMgetnextarrtruentry(varr, indices));
                                            /* Get the next index tuple */
  return 1;
}

int main() {
  int NSCENAR = 5;
  const char *DEMDAT[]={"dem1.dat", "dem2.dat", "dem3.dat", "dem4.dat", "dem5.dat"};
  double WEIGHT[] = {0.4, 0.15, 0.1, 0.25, 0.1};
  int i,s;
  double fact = 0;

  if(XPRMinit()) return 1;                  /* Initialize Mosel */

  for(i=0;i<NSCENAR;i++) if(execscenar(DEMDAT[i], WEIGHT[i])) fact+=WEIGHT[i];
  fact= 1/fact;

  printf("Weighted average objective:  %g\n", obj*fact);
  for(s=0;s<NSUP;s++)                       /* Print capacity use forecasts */
    printf("%s:  %g (%g%% of %g)\n", SNAME[s], capuse[s]*fact,
      capuse[s]*fact/CAP[s]*100, CAP[s]);
  return 0;
}
```

**Figure 4**: Executing a Mosel model with several data sets and analysis of the results

 ©2009 Fair Isaac Corporation. All rights reserved. 

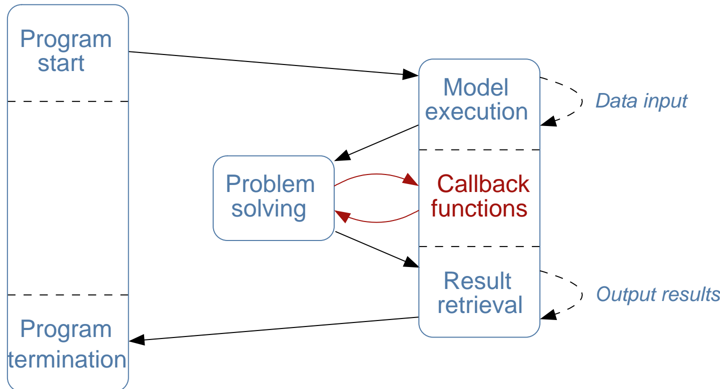


**Figure 5**: Interaction with the solution algorithm

$$\forall a \in AREAS : \sum_{c \in CONTR} alloc_{ca} \leq UPPCON_a$$

Furthermore, we need to establish the relation between the two sets of variables: a contractor *c* is allocated to an area *a* if and only if he is allocated a site *s* in this area, that is, $y_{ca}$ is 1 if and only if some $x_{cs}$ (for a site *s* in area *a*) is 1. This equivalence is expressed by the following two sets of constraints, one for each sense of the implication ($AREA_s$ is the area a site *s* belongs to and $NUMS_a$ the number of sites in area *a*):

$$\forall c \in CONTR, a \in AREAS : alloc_{ca} \leq \sum_{\substack{s \in SITES \\ AREA_s = a}} clean_{cs} \tag{3.1}$$

$$\forall c \in CONTR, a \in AREAS : alloc_{ca} \geq \frac{1}{NUMS_a} \cdot \sum_{\substack{s \in SITES \\ AREA_s = a}} clean_{cs} \tag{3.2}$$

The second set of constraints (3.2) is an aggregated formulation (so-called Multiple Variable Lower Bound, MVLB, constraints) that has a relatively "weak" LP-relaxation. A stronger formulation of the same, with a considerably larger number of constraints, is given by the following:

$$\forall c \in CONTR, s \in SITES : alloc_{c,AREA_s} \geq clean_{cs} \tag{3.3}$$

Since the constraints (3.2) are sufficient to state the problem, we can use the constraints (3.3) to define a branch-and-cut algorithm that performs the following steps at every node in the branch-and-bound search tree:

- Solve the LP-relaxation.
- Get the solution values.
- Identify violated constraints of type (3.3).
- Add the violated inequalities to the problem.
- Re-solve the LP.

## 3.2 Implementation

A Mosel implementation of the cut generation function[1] is given in Figure 6, using Xpress-Optimizer to solve the problem.

[1] The complete program is given in the appendix.


©2009 Fair Isaac Corporation. All rights reserved.

```
public function cb_node:boolean
  declarations
    ncut:  integer                     !  Counters for cuts
    cut:  array(range) of linctr       !  Cuts
    cutid:  array(range) of integer    !  Cut type identification
    type:  array(range) of integer     !  Cut constraint type
  end-declarations

  returned:=false                      !  Call this function once per node

  depth:=getparam("XPRS_NODEDEPTH")

  if(depth<4) then
    ncut:=0

    !  Get the solution values
    forall(c in CONTR) do
      forall(a in AREAS) sola(c,a):=getsol(alloc(c,a))
      forall(s in SITES) solc(c,s):=getsol(clean(c,s))
    end-do

    !  Search for violated constraints
    forall(c in CONTR, s in SITES)
      if(solc(c,s)-sola(c,AREA(s)) > 0) then
        cut(ncut):= alloc(c,AREA(s)) - clean(c,s)
        cutid(ncut):= 1
        type(ncut):= CT_GEQ
        ncut+=1
      end-if

    !  Add cuts to the problem
    if(ncut>0) then
      addcuts(cutid, type, cut)
      returned:=true                   !  Call this function again
    end-if
  end-if

end-function
```

**Figure 6**: Cut generation function

In this implementation we restrict the generation of cuts to the first three levels, *i.e.* `depth < 4`, of the search tree. The procedure `addcuts` requires the following information for every cut: a user defined identifier (sometimes useful if we want to refer to the cuts later; here we simply assign the value '1' to all cuts), the type of the relation (here `CT_GEQ`, that is $\geq$), and the cut terms in normalized form (all variable and constant terms on the left hand side of the inequality). The three arrays storing this information grow dynamically each time a new cut is added. At every node the cut generation function is called repeatedly, followed by a re-solution of the current LP, as long as it returns `true`.

The definition of this function needs to be communicated to the solver before the problem is solved. There are also a few paramter settings, and some declarations that are made in the main model for efficiency reasons. In addition, we define a parameter `ALG` to select the solution algorithm when executing the model (in stand-alone mode or from an application as shown in the previous section). Figure 7 shows the corresponding Mosel program extract.

## 3.3 Results

A question that may come to mind when writing (parts of) algorithms in the Mosel language is whether there is any substantial loss in terms of execution speed. We have run our model of the Office Cleaning problem with a real-world data set with 4 contractors and 4 areas with a total of 562 sites. The model has about 1,700 binary variables and 600 constraints. Xpress-Optimizer with default settings but without any cut generation (`ALG=2`) takes 22 nodes and 0.28 seconds (on a Pentium 3, 550MHz), and with default cut generation and a sufficiently large number of spare matrix rows available (`ALG=3`) the problem is solved at the root node in 0.74 seconds. With our cut generation strategy (`ALG=1`) it takes 5 nodes and 0.44 seconds to solve the problem and in total over 650 cuts are added to the matrix.


©2009 Fair Isaac Corporation. All rights reserved.

```
parameters
  ALG = 0                                     !  Default algorithm:  no user cuts
end-parameters

declarations    feastol:  real                !  Zero tolerance
  solc:  array(CONTR,SITES) of real           !  Sol.  values for variables 'clean'
  sola:  array(CONTR,AREAS) of real           !  Sol.  values for variables 'alloc'
end-declarations

case ALG of
 1:  do
      setparam("XPRS_CUTSTRATEGY", 0)         !  Disable automatic cuts
      setparam("XPRS_HEURSTRATEGY", 0)        !  Switch MIP heuristics off
      setparam("XPRS_PRESOLVE", 0)            !  Switch presolve off
      setparam("XPRS_EXTRAROWS", 5000)        !  Reserve extra rows in matrix
      feastol:= getparam("XPRS_FEASTOL")      !  Get the zero tolerance
      setparam("zerotol", feastol * 10)       !  Set Mosel comparison tolerance
      setcallback(XPRS_CB_CM, "cb_node")      !  Define the cut generation callback
    end-do
 2:  setparam("XPRS_CUTSTRATEGY", 0)          !  No cut generation
 3:  setparam("XPRS_EXTRAROWS", 5000)         !  Limit the number of cuts
end-case

minimize(Cost)                                !  Solve the MIP problem
```

**Figure 7**: Setup for cut generation

# 4 Column generation for cutting-stock

The last case we consider is the interaction between the Mosel model and the calling program. A typical issue is the communication in-memory of data available in the calling program to a Mosel model (Figure 8), that is, without having to write them out to files. Another issue is the access to intermediate results during the execution of a model.

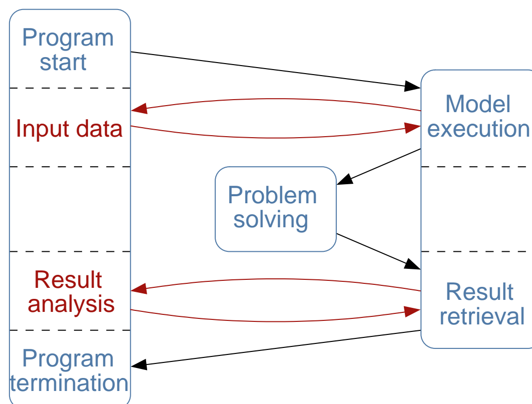


**Figure 8**: Interaction with the calling program

The implementation of a cutting-stock problem described next illustrates two points:

- data input in memory and result retrieval during model execution and
- the definition of a column generation algorithm.

## 4.1 Cutting-stock problem

A paper mill produces rolls of paper of a fixed width *MAXWIDTH* that are subsequently cut into smaller rolls according to the customer orders. The rolls can be cut into *NWIDTHS* different sizes. The orders are given as demands for each width $i$ ($DEMAND_i$). The objective of the paper mill is


©2009 Fair Isaac Corporation. All rights reserved.

to satisfy the demand with the smallest possible number of paper rolls in order to minimize the losses.

We express the objective of minimizing the total number of rolls as choosing the best set of cutting patterns for the current set of demands. To avoid having to calculate all possible cutting patterns by hand, we start off with a basic set of patterns ($PATTERN_1$,..., $PATTERN_{NWIDTH}$), that consists of cutting small rolls all of the same width as many times as possible out of the large roll. Let variables $use_j$ denote the number of times a cutting pattern $j$ ($j \in WIDTHS = \{1, \ldots, NWIDTH\}$) is used. The objective then becomes to minimize the sum of these variables, subject to the constraints that the demand for every size has to be met. (Function *ceil* means rounding to the next larger integer value.)

$$\text{minimize} \sum_{j \in WIDTHS} use_j$$

$$\sum_{j \in WIDTHS} PATTERN_{ij} \cdot use_j \geq DEMAND_i$$

$$\forall j \in WIDTHS : use_j \leq \text{ceil}(DEMAND_j \,/\, PATTERN_{jj}), \quad use_j \text{ integer}$$

## 4.2 Implementation

### 4.2.1 Mosel model

As before, the transformation of the mathematical model into a Mosel program (Figure 9) is fairly straightforward. However, the implementation uses some specific features:

**Decision variables `use`:** The index range is not known at the start of the program, the array of variables will grow dynamically during the column generation.

**Constraints:** Here all the constraints are named for further reference (we need to add new terms during the column generation).

**Data:** The arrays `WIDTH` and `DEMAND` are initialized with values from the calling program (see Figure 15 below). The data are obtained by calling the procedure `getcutstkdata`, defined by the module `cutstkdata`.

The column generation algorithm (see Figures 16 and 10) is implemented as a loop over the following four steps that is repeated as long as new cutting patterns are generated:

1. Solve the LP and save the basis.
2. Get the solution values.
3. Generate a new column (=cutting pattern) by solving a knapsack problem on the dual values.
4. Load the modified problem and load the saved basis.

Notice that the knapsack problem solved to generate a new column is an independent optimization problem that is defined and solved in the same Mosel model as the main cutting stock problem[2].

Re-starting the solution of the LP problem with the basis of the previous optimal solution reduces the number of simplex iterations required to solve the problem and hence leads to shorter running times. Mosel completes the saved basis according to the additions made to the problem.

[2] The complete program is given in the appendix.


©2009 Fair Isaac Corporation. All rights reserved.

```
model Papermill
  uses "mmxprs"                                   !  Use Xpress-Optimizer
  uses "cutstkdata"                               !  User module

  parameters
    NWIDTHS = 5                                   !  Number of different widths
    WDATA=''                                      !  Location of width data in memory
    WSIZE=0                                       !  Size of the data block
    DDATA=''                                      !  Location of demand data in memory
    DSIZE=0                                       !  Size of the data block
  end-parameters

  forward procedure column_gen
  !  Solve the problem z = max{cx :  ax<=b, x in Z^N} with N=NWIDTHS :
  forward function knapsack(C:array(range) of real, A:array(range) of real,
                            B:real, xbest:array(range) of integer, pass:integer):  real

  declarations
    WIDTHS = 1..NWIDTHS                           !  Range of widths
    RP: range                                     !  Range of cutting patterns
    MAXWIDTH = 94                                 !  Maximum roll width
    EPS = 0.001                                   !  Zero tolerance
    WIDTH: array(WIDTHS) of real                  !  Possible widths
    DEMAND: array(WIDTHS) of integer              !  Demand per width
    PATTERN: array(WIDTHS, WIDTHS) of integer     !  (Basic) cutting patterns

    use:  array(RP) of mpvar                      !  Rolls per pattern
    DemC: array(WIDTHS) of linctr                 !  Demand constraints
    MinRolls:  linctr                             !  Objective function
  end-declarations

  getcutstkdata(WIDTH, WDATA, WSIZE)              !  Get width data
  getcutstkdata(DEMAND, DDATA, DSIZE)             !  Get demand data

                                                  !  Make basic patterns
  forall(j in WIDTHS) PATTERN(j,j):= floor(MAXWIDTH/WIDTH(j))

  forall(j in WIDTHS) do
    create(use(j))                                !  Create NWIDTHS variables 'use'
    use(j) is_integer                             !  Variables are integer and bounded
    use(j) <= integer(ceil(DEMAND(j)/PATTERN(j,j)))
  end-do
  MinRolls:= sum(j in WIDTHS) use(j)              !  Objective:  minimize no.  of rolls
  forall(i in WIDTHS)                             !  Satisfy all demands
  DemC(i):= sum(j in WIDTHS) PATTERN(i,j) * use(j) >= DEMAND(i)

  column_gen                                      !  Column generation algorithm
  minimize(MinRolls)                              !  Solve the resulting problem
end-model
```

**Figure 9**: Cutting stock model

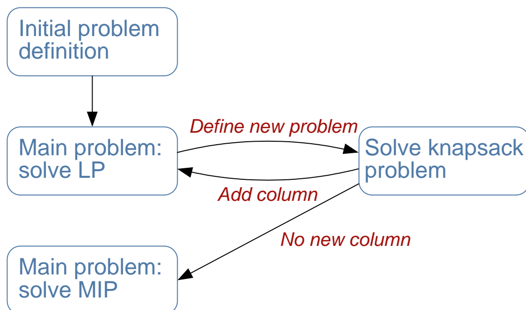


**Figure 10**: Column generation algorithm


©2009 Fair Isaac Corporation. All rights reserved.

```
procedure column_gen
  declarations
    dualdem:  array(WIDTHS) of real
    xbest:  array(WIDTHS) of integer
    ubnd, zbest:  real
    bas:  basis
  end-declarations

  setparam("XPRS_CUTSTRATEGY", 0)          !  Disable automatic cuts
  setparam("XPRS_PRESOLVE", 0)             !  Switch presolve off
  npatt:=NWIDTHS
  npass:=1

  while(true) do
    minimize(XPRS_LIN, MinRolls)           !  Solve the LP
    savebasis(bas)                         !  Save the current basis
                                           !  Get the dual values
    forall(i in WIDTHS) dualdem(i):=getdual(DemC(i))
                                           !  Solve a knapsack problem
    zbest:= knapsack(dualdem, WIDTH, MAXWIDTH, xbest, npass) - 1.0

    if(zbest < EPS) then
      writeln("No profitable column found.\n")
      break
    else
      printpat(zbest, xbest, WIDTH)        !  Print the new pattern
      npatt+=1
      create(use(npatt))                   !  Create a variable for this pattern
      use(npatt) is_integer
      MinRolls+= use(npatt)                !  Add new var.  to the objective
      ubnd:=0
      forall(i in WIDTHS)
        if(xbest(i) > 0) then
          DemC(i)+= xbest(i)*use(npatt)   !  Add new var.  to demand constraints
          ubnd:= maxlist(ubnd, ceil(DEMAND(i)/xbest(i) ))
        end-if
      use(npatt) <= ubnd                   !  Set upper bound on the new variable

      loadprob(MinRolls)                   !  Reload the problem
      loadbasis(bas)                       !  Load the saved basis
    end-if
    npass+=1
  end-do
end-procedure
```

**Figure 11**: Column generation algorithm

 ©2009 Fair Isaac Corporation. All rights reserved. 

```
#include <stdio.h>
#include <stdlib.h>
#include "xprm_mc.h"
#include "xprm_ni.h"

/* Initialization function of the module 'cutstkdata' */
static int cutstkdata_init(XPRMnifct nifct, int *interver,int *libver, XPRMdsointer **interf);

int main()
{
  XPRMmodel mod;
  int result;
  char params[80];
  static int tabdemand[] = 150, 96, 48, 108, 227;
  static double tabwidth[] =17, 21, 22.5, 24, 29.5;
  int nwidths = 5;

/* Initialize Mosel */
  if(XPRMinit()) return 1;

/* Register 'cutstkdata' as a static module (=stored in the program) */
  if(XPRMregstatdso("cutstkdata", cutstkdata_init)) return 2;

/* Parameters:  the addresses of the data tables and their sizes */
  sprintf(params, "DDATA='%p',DSIZE=%d,WDATA='%p',WSIZE=%d,NWIDTHS=%d",
          tabdemand, sizeof(tabdemand)/sizeof(int), tabwidth,
          sizeof(tabwidth)/sizeof(double), nwidths);

/* Execute the model */
  if(XPRMexecmod("", "paperm.mos", params, &result, &mod)) return 3;

  return result;
}
```

**Figure 12**: C program: initialization and model execution

### 4.2.2 C program

The execution of this model (`paperm.mos`) is started from the following C program. Before executing the model we need to declare the *module* `cutstkdata` that is used by it. This module is *static* because it is included in the C program that executes the model. If a module is not specific to an application (e.g. a solver module), it should be compiled into a *Dynamic Shared Object* that can be used by any model independent of the way it is executed.

For more detail on the correct formating of the initialization function and the interface structures the reader is refered to the software documentation [4]. With Xpress-IVE the corresponding C code can be generated automatically, including templates for the library functions.

Figure 13 shows the module function implementing the procedure `printpat`. In addition to this function, the module also defines the two versions of the procedure `setcutstkdata` used in Figure 9, for integer and real-valued data arrays. A module may *overload* any subroutines (including those defined by Mosel), provided that every version of the subroutine has a different number or type(s) of arguments.

Certain similarities with the example in Section 2 may be observed, but here we interact with a model during its execution: instead of the Mosel Library functions we need to use the Native Interface functions (stored in the structure `mm` that is defined by the module initialization function) and work directly with the Mosel stack (using `XPRM_POP_`... to take objects from the stack).

## 4.3 Results for bin packing benchmark problems

We have tried the column generation heuristic with several sets of bin packing benchmark problems.

To speed up the implementation, a greedy-type heuristic (based on the LPT heuristic [6]) has been added before the start of the column generation: at every iteration the heuristic places the largest unassigned item into the bin (=roll) with the least load. The number of bins is initially set

 ©2009 Fair Isaac Corporation. All rights reserved. 

```
static XPRMnifct mm;                          /* To store Mosel's function table */
static int printpattern(XPRMcontext ctx,void *libctx)
{
  XPRMarray varr, warr;
  int v, index[1];
  double dj, w, tw=0;
  dj=XPRM_POP_REAL(ctx);
  varr=XPRM_POP_REF(ctx);                     /* The value array */
  warr=XPRM_POP_REF(ctx);                     /* The widths array */
  mm->printf(ctx, "New pattern found with marginal cost %g\n", dj);
  mm->printf(ctx, " Widths distribution:  ");
  mm->getfirstarrentry(varr, index);          /* Get the first index tuple */
  do {
    mm->getarrval(varr, index, &v);           /* Get the value */
    mm->getarrval(warr, index, &w);           /* Get the width */
    mm->printf(ctx, "%g:%d ", w, v);
    tw += v*w;
  } while(!mm->getnextarrentry(varr, index));
  mm->printf(ctx, "Total width:  %g\n", tw);
  return XPRM_RT_OK;
}
```

**Figure 13**: Library function implementing the subroutine `printpat`

to the theoretical lower bound (ceil($\sum_{i \in WIDTHS} DEMAND_i \cdot WIDTH_i \, / \, MAXWIDTH$)). Whenever an item no longer fits into the existing bins, a new bin is added. The patterns (=distribution of items to bins) generated by this heuristic are completed to non-dominated patterns and added as new columns to the problem. Some instances are solved immediately by this heuristic, otherwise the new columns provide an improved starting point for the MIP-based column generation.

The performance of our algorithm clearly depends on the characteristics of the problems: the benchmark problems are classified according to the number of items (between 50 and 1000) that need to be placed into bins of a fixed size. However, the total number of widths is a more suitable measure of difficulty in our case: for up to about 100 different widths the algorithm is very efficient, taking on the average only a few seconds. In most cases the optimal solution is found, but sometimes the best solution found is 1-2 rolls (bins) worse than the optimum (mainly due to the tolerance and time limit on the knapsack problem, and probably also to rounding errors). For six instances where the optimal solution was previously not known, our algorithm has found the optimal solution or improved the value of the best solution.

For instances with a larger number of widths (up to 350) the results summarised in Tables 1 and 2 are often 5-10 rolls (bins) worse than the best solution. They can be improved with longer running times (e.g. 10 seconds per knapsack problem and 1000 seconds per main MIP problem), but specialized heuristics may be a better choice. Especially for problems like the t*xxx* series that has been constructed as triplets adding up to the size of the bins a heuristic exploiting the specific problem structure is likely to outperform our results. Other factors influencing the results are the bin size (= maximum width) and the relative size of the items. Many instances that are hard for our algorithm have widths in a relatively small range varying around 1/5 to 1/3 of the bin size.

The results reported in Tables 1 and 2 have been obtained by limiting the solving time per knapsack problem to 2 seconds and the time for solving the MIP of the main problem to 2 minutes. There was no limit on the number of patterns (= new columns) generated. The execution time of the heuristic and the overhead for re-loading the problems into Xpress-Optimizer are negligible compared to the solving times: even for the largest instances the initial heuristic takes less than 0.05 seconds. For one of the largest instances (350 widths, evolving from 534 to 1246 patterns) more than 700 column generation iterations are performed in 450 seconds of which only 3.8 seconds are spent for the more than 1400 re-loadings of problems. The times are measured on a PC P4, 1.5GHz, 512Mb.

The heuristic works well for the instances in Table 1. For the instances in Table 2 it never finds the best solution, but usually it helps speed up the column generation.


©2009 Fair Isaac Corporation. All rights reserved.

| | Instance | | Results | | | | | | Heuristic | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Name** | **Items** | **Widths** | **Time** | **Dev.** | **Eql.** | **-1** | **+1** | **>+1** | **Eql.** | **Stop** |
| **N1CW1** | 50 | 39.18 | 1.00 | 0.10 | 54 | – | 6 | – | 43 | 6 |
| **N1CW2** | 50 | 37.40 | 2.20 | 0.03 | 58 | – | 2 | – | 35 | 2 |
| **N1CW4** | 50 | 36.02 | 0.89 | 0.15 | 52 | – | 7 | 1 | 44 | – |
| **N2CW1** | 100 | 63.02 | 2.99 | 0.23 | 46 | – | 14 | – | 25 | – |
| **N2CW2** | 100 | 57.45 | 7.31 | 0.03 | 58 | – | 2 | – | 37 | – |
| **N2CW4** | 100 | 53.32 | 6.24 | 0.10 | 54 | – | 6 | – | 39 | – |
| **N3CW1** | 200 | 86.07 | 10.33 | 0.22 | 47 | – | 13 | – | 15 | – |
| **N3CW2** | 200 | 74.60 | 8.81 | 0.07 | 56 | – | 4 | – | 37 | – |
| **N3CW4** | 500 | 67.07 | 13.02 | 0.03 | 58 | – | 2 | – | 36 | – |
| **N4CW1** | 500 | 99.52 | 46.30 | 0.72 | 37 | – | 8 | 15 | 13 | – |
| **N4CW2** | 500 | 80.83 | 9.30 | 0.12 | 53 | – | 7 | – | 37 | – |
| **N4CW4** | 500 | 70.93 | 17.82 | -0.03 | 54 | 4 | 2 | – | 40 | – |
| **N1W1B** | 50 | 45.80 | 22.97 | 0.27 | 22 | – | 8 | – | 13 | 8 |
| **N1W2B** | 50 | 42.67 | 1.74 | 0.23 | 24 | – | 5 | 1 | 30 | 24 |
| **N1W3B** | 50 | 40.77 | 1.82 | 0.27 | 22 | – | 8 | – | 30 | 22 |
| **N1W4B** | 50 | 38.53 | 0.55 | 0.07 | 28 | – | 2 | – | 30 | 28 |
| **N2W1B** | 100 | 82.60 | 79.54 | 0.60 | 15 | – | 12 | 3 | 7 | – |
| **N2W2B** | 100 | 73.87 | 28.40 | 0.77 | 12 | – | 14 | 4 | 30 | 12 |
| **N2W3B** | 100 | 66.47 | 5.31 | 0.23 | 23 | – | 7 | – | 30 | 23 |
| **N2W4B** | 100 | 61.47 | 7.59 | 0.40 | 18 | – | 12 | – | 30 | 18 |
| **N3W1B** | 200 | 140.60 | 336.82 | 1.67 | 3 | – | 9 | 18 | 3 | – |
| **N3W2B** | 200 | 117.17 | 119.76 | 1.43 | 7 | – | 13 | 10 | 30 | 7 |
| **N3W3B** | 200 | 99.50 | 22.67 | 0.33 | 20 | – | 10 | – | 30 | 20 |
| **N3W4B** | 200 | 86.07 | 17.10 | 0.57 | 17 | – | 9 | 4 | 30 | 17 |
| **N4W1B** | 500 | 241.93 | 625.94 | 9.20 | 1 | – | – | 29 | 10 | – |
| **N4W2B** | 500 | 177.17 | 341.66 | 5.77 | – | – | 1 | 29 | 24 | – |
| **N4W3B** | 500 | 135.57 | 156.25 | 1.07 | 1 | – | 13 | 16 | 30 | 1 |
| **N4W4B** | 500 | 111.13 | 52.55 | 0.63 | 14 | – | 10 | 6 | 30 | 14 |

**Table 1**: Test instances by Scholl/Klein/Jürgens [9]. First set 60, second set 30 instances per line.

| | Instance | | Results | | | | | | Heuristic | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Name** | **Items** | **Widths** | **Time** | **Dev.** | **Eql.** | **-1** | **+1** | **>+1** | **Eql.** | **Stop** |
| **u120** | 120 | 63.20 | 22.97 | 0 | 18 | 1 | 1 | – | – | 0 |
| **u240** | 240 | 77.25 | 30.40 | -0.05 | 19 | 1 | – | – | – | 0 |
| **u500** | 500 | 80.80 | 26.77 | 0 | 20 | – | – | – | – | 0 |
| **u1000** | 1000 | 81.00 | 68.61 | 0.3 | 14 | – | 6 | – | – | 0 |
| **t60** | 60 | 49.95 | 30.61 | 0.95 | 2 | – | 17 | 1 | 3 | 0 |
| **t120** | 120 | 86.15 | 84.63 | 1.1 | – | – | 19 | 1 | 1 | 0 |
| **t249** | 249 | 140.10 | 258.81 | 2.6 | – | – | 5 | 15 | 2 | 0 |
| **t501** | 501 | 194.25 | 308.30 | 10 | – | – | – | 20 | 3 | 0 |

**Table 2**: Test instances from Beasley's OR Library [1]. Both sets 20 instances per line.


©2009 Fair Isaac Corporation. All rights reserved.

## 4.4 Simplified data exchange in memory using I/O drivers

The generalization of the notion 'file' and the introduction of *I/O drivers* in Mosel replace certain uses of static user modules. In particular for transfering data in memory it is often no longer necessary to write a dedicated module.

The input of data at the beginning of the Mosel model is modified as follows.

```
model Papermill
  uses "mmxprs"                           ! Use Xpress-Optimizer

  parameters
    NWIDTHS = 5                           ! Number of different widths
    WDATA=''                              ! Location of width data in memory
    DDATA=''                              ! Location of demand data in memory
  end-parameters

  ...

  declarations
    WIDTHS = 1..NWIDTHS                   ! Range of widths
    ...
    WIDTH: array(WIDTHS) of real          ! Possible widths
    DEMAND: array(WIDTHS) of integer      ! Demand per width
    ...
  end-declarations

  initializations from "raw:"
    WIDTH as WDATA
    DEMAND as DDATA
  end-initializations

  ...
end-model
```

**Figure 14**: Cutting stock model using I/O drivers

The C program now is considerably shorter as before, even if we add retrieval of the solution from the model after its execution.

 ©2009 Fair Isaac Corporation. All rights reserved. 

```
#include <stdio.h>
#include <stdlib.h>
#include "xprm_mc.h"

int main()
{
  XPRMmodel mod;
  XPRMalltypes rvalue;
  XPRMmemblk *memblk;
  int result;
  char params[200];
  static int tabdemand[] = 150, 96, 48, 108, 227;
  static double tabwidth[] =17, 21, 22.5, 24, 29.5;
  double *tabsol;
  int i, type, sizesol;
  int nwidths = 5;

/* Initialize Mosel */
  if(XPRMinit()) return 1;

/* Parameters:  the addresses of the data tables and their sizes */
  sprintf(params, "DDATA='noindex,mem:%#lx/%u',WDATA='noindex,mem:%#lx/%u',NWIDTHS=%d",
         (unsigned long)tabdemand, sizeof(tabdemand), (unsigned long)tabwidth,
         sizeof(tabwidth), nwidths);

/* Execute the model */
  if(XPRMexecmod("", "paperm.mos", params, &result, &mod)) return 2;

/* Test whether a solution is found */
  if((XPRMgetprobstat(mod)&XPRM_PBRES)!=XPRM_PBOPT) return 3;

/* Retrieve solution data */
  type=XPRMfindident(mod, "soltab", &rvalue);
  if(XPRM_STR(type)!=XPRM_STR_MEM) return 4;

  memblk=rvalue.memblk;
  tabsol=(double *)(memblk->ref);
  sizesol=(int)(memblk->size/sizeof(double));

/* Print out the solution */
  printf("Best integer solution:  %g rolls\n", XPRMgetobjval(mod));
  printf(" Rolls per pattern:  ");
  for(i=0;i<sizesol;i++) printf("%g, ", tabsol[i]);
  printf("\n");

  return result;
}
```

**Figure 15**: Complete C program using I/O drivers for data exchange

The definition of the procedure `printpat` has been moved to the Mosel model:

```
procedure printpat(zbest:real, xbest:array(range) of integer,
                   W:array(range) of real)
  declarations
    dw:  real
  end-declarations

  writeln("New pattern found with marginal cost ", zbest)
  write(" Widths distribution:  ")
  dw:=0
  forall(i in WIDTHS) do
    write(W(i), ":", xbest(i), " ")
    dw += W(i)*xbest(i)
  end-do
  writeln("Total width:  ", dw)
end-procedure
```

**Figure 16**: Prniting of new patterns

# 5 Conclusion

The algorithms implemented in Mosel described in this paper give the reader an idea of the flexibility of the Mosel language. With Mosel, the user may loop over optimization statements, implement solution heuristics (self-contained or involving external solvers) and handle input and output in a transparent way.


©2009 Fair Isaac Corporation. All rights reserved.

Three different types of embedding into and interacting with external application programs are shown with the help of the same examples: "traditional embedding", interaction with the solver, and interaction with the calling program.

The examples in this paper go far beyond typical textbook modeling examples: our purpose is to show how complicated algorithms can be implemented and embedded efficiently. In many cases there will be no need for anything more than the "traditional embedding". However, if for instance, additions to the problem definition or an increase in size render a problem impractical to solve with standard solution approaches, the possibility to continue working in the same modeling environment whilst experimenting with new solution strategies may lead to a substantial gain in application development time.

# Bibliography

[1] J. E. Beasley. `http://mscmga.ms.ic.ac.uk/jeb/orlib/binpackinfo.html`

[2] T. A. Ciriani, Y. Colombani and S. Heipcke. Optimization Modeling Innovations. Dash Optimization White Paper, 2002.

[3] Y. Colombani and S. Heipcke. Mosel: An Extensible Environment for Modeling and Programming Solutions. In N. Jussien and F. Laburthe, editors, *Proceedings of CP-AI-OR'02*, pages 277–290, Le Croisic, March 2002.

[4] Dash Optimization - part of Fair Issac. *Mosel Native Interface Reference Manual*. Blisworth, UK, 2002.

[5] R. Fourer, D. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.

[6] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *Siam Journal on Applied Mathematics*, 17:416–429, 1969.

[7] S. Heipcke. *Combined Modelling and Problem Solving in Constraint Programming and Mathematical Programming*. PhD thesis, University of Buckingham, 1999.

[8] S. Heipcke. *Applications of Optimization with Xpress-MP*. Dash Optimization, Blisworth, UK, 2002.

[9] A. Scholl, R. Klein, and C. Jürgens. `http://www.bwl.tu-darmstadt.de/bwl3/forsch/projekte/binpp/`.

[10] J. R. Tebboth and R. C. Daniel. A Tightly Integrated Modelling and Optimisation Library. *Annals of Operations Research*, 104:313–333, 2001.

[11] P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA, 1998.


©2009 Fair Isaac Corporation. All rights reserved.

# Appendix: code listings

## Office cleaning problem

Mosel file `cleana.mos`:


©2009 Fair Isaac Corporation. All rights reserved.

```
model "Office cleaning"
  uses "mmxprs","mmsystem"

  parameters
    ALG = 0                                     ! Default algorithm:  no user cuts
  end-parameters

  forward public function cb_node:boolean

  declarations
    PARAM: array(1..3) of integer
  end-declarations

  initializations from 'cldata.dat'
    PARAM
  end-initializations

  declarations
    NSITES = PARAM(1)                           ! Number of sites
    NAREAS = PARAM(2)                           ! Number of areas (subsets of sites)
    NCONTRACTORS = PARAM(3)                     ! Number of contractors
    AREAS = 1..NAREAS
    CONTR = 1..NCONTRACTORS
    SITES = 1..NSITES
    AREA: array(SITES) of integer               ! Area site is in
    NUMSITE: array(AREAS) of integer            ! Number of sites in an area
    LOWCON: array(AREAS) of integer             ! Lower limit on the number of
                                                ! contractors per area
    UPPCON: array(AREAS) of integer             ! Upper limit on the number of
                                                ! contractors per area
    ADJACENT: array(AREAS,AREAS) of integer     ! 1 if areas adjacent, 0 otherwise
    PRICE: array(SITES,CONTR) of real           ! Price per contractor per site

    clean:  dynamic array(CONTR,SITES) of mpvar ! 1 iff contractor c cleans site s
    alloc:  array(CONTR,AREAS) of mpvar         ! 1 iff contractor allocated to a site
                                                ! in area a

    feastol:  real                              ! Zero tolerance
    solc:  array(CONTR,SITES) of real           ! Sol.  values for variables 'clean'
    sola:  array(CONTR,AREAS) of real           ! Sol.  values for variables 'alloc'
  end-declarations

  initializations from 'cldata.dat'
    [NUMSITE,LOWCON,UPPCON] as 'AREADATA'
    ADJACENT
    PRICE
  end-initializations

  ct:=1
  forall(a in AREAS) do
    forall(s in ct..ct+NUMSITE(a)-1)
      AREA(s):=a
    ct+= NUMSITE(a)
  end-do

  forall(c in CONTR, s in SITES | PRICE(s,c) > 0.01) create(clean(c,s))

!  Objective:  Minimize total cost of all cleaning contracts
  Cost:= sum(c in CONTR, s in SITES) PRICE(s,c)*clean(c,s)

!  Each site must be cleaned by exactly one contractor
  forall(s in SITES) sum(c in CONTR) clean(c,s) = 1

!  Ban same contractor from serving adjacent areas
  forall(c in CONTR, a,b in AREAS | a > b and ADJACENT(a,b) = 1)
    alloc(c,a) + alloc(c,b) <= 1

!  Specify lower & upper limits on contracts per area
  forall(a in AREAS | LOWCON(a)>0)
    sum(c in CONTR) alloc(c,a) >= LOWCON(a)
  forall(a in AREAS | UPPCON(a)<NCONTRACTORS)
    sum(c in CONTR) alloc(c,a) <= UPPCON(a)

!  Define alloc(c,a) to be 1 iff some clean(c,s)=1 for sites s in area a
  forall(c in CONTR, a in AREAS) do
    alloc(c,a) <= sum(s in SITES| AREA(s)=a) clean(c,s)
    alloc(c,a) >= 1.0/NUMSITE(a) * sum(s in SITES| AREA(s)=a) clean(c,s)
  end-do

  forall(c in CONTR) do
    forall(s in SITES) clean(c,s) is_binary
    forall(a in AREAS) alloc(c,a) is_binary
  end-do
```

 ©2009 Fair Isaac Corporation. All rights reserved. 

```
  starttime:= gettime

  case ALG of
   1:  do
        setparam("XPRS_CUTSTRATEGY", 0)      !  Disable automatic cuts
        setparam("XPRS_HEURSTRATEGY", 0)     !  Switch MIP heuristics off
        setparam("XPRS_PRESOLVE", 0)         !  Switch presolve off
        setparam("XPRS_EXTRAROWS", 5000)     !  Reserve extra rows in matrix
        feastol:= getparam("XPRS_FEASTOL")   !  Get the zero tolerance
        setparam("zerotol", feastol * 10)    !  Set Mosel comparison tolerance
        setcallback(XPRS_CB_CM, "cb_node")   !  Define the cut generation callback
      end-do
   2:  setparam("XPRS_CUTSTRATEGY", 0)       !  No cuts
   3:  setparam("XPRS_EXTRAROWS", 5000)      !  Limit the number of cuts
  end-case

  minimize(Cost)                             !  Solve the MIP problem
  writeln("(", gettime-starttime, " sec) Global status ",
        getparam("XPRS_MIPSTATUS"), ", best solution:  ", getobjval);

!  ****Cut generation loop at tree nodes****
  public function cb_node:boolean
    declarations
      ncut:  integer                         !  Counters for cuts
      cut:  array(range) of linctr           !  Cuts
      cutid:  array(range) of integer        !  Cut type identification
      type:  array(range) of integer         !  Cut constraint type
    end-declarations

    returned:=false                          !  Call this function once per node

    depth:=getparam("XPRS_NODEDEPTH")

    if(depth<4) then
      ncut:=0

      !  Get the solution values
      forall(c in CONTR) do
        forall(a in AREAS) sola(c,a):=getsol(alloc(c,a))
        forall(s in SITES) solc(c,s):=getsol(clean(c,s))
      end-do

      !  Search for violated constraints
      forall(c in CONTR, s in SITES)
        if(solc(c,s)-sola(c,AREA(s)) > 0) then
          cut(ncut):= alloc(c,AREA(s)) - clean(c,s)
          cutid(ncut):= 1
          type(ncut):= CT_GEQ
          ncut+=1
        end-if

      !  Add cuts to the problem
      if(ncut>0) then
        addcuts(cutid, type, cut)
        returned:=true                       !  Call this function again
      end-if
    end-if

  end-function
end-model
```

## C program:

```
#include <stdio.h>
#include "xprm_mc.h"

int main()
{
  XPRMmodel mod;
  int result;

  if(XPRMinit()) return 1;   /* Initialize Mosel */
                             /* Execute the model with different algorithms */
  if(XPRMexecmod(NULL, "cleana.mos", NULL, &result, &mod)) return 2;
  if(XPRMexecmod(NULL, "cleana.mos", "ALG=1", &result, &mod)) return 3;
  if(XPRMexecmod(NULL, "cleana.mos", "ALG=2", &result, &mod)) return 4;
  if(XPRMexecmod(NULL, "cleana.mos", "ALG=3", &result, &mod)) return 5;

  return 0;
}
```


©2009 Fair Isaac Corporation. All rights reserved.

## Cutting-stock Problem

### Mosel file paperm.mos:

```
model Papermill   uses "mmxprs"                         ! Use Xpress-Optimizer
  uses "cutstkdata"                                     ! User module

  parameters
    NWIDTHS = 5                                         ! Number of different widths
    WDATA=''                                            ! Location of width data in memory
    WSIZE=0                                             ! Size of the data block
    DDATA=''                                            ! Location of demand data in memory
    DSIZE=0                                             ! Size of the data block
  end-parameters

  forward procedure column_gen
  forward function knapsack(C:array(range) of real, A:array(range) of real,
                            B:real, xbest:array(range) of integer, pass:integer):  real

  declarations
    WIDTHS = 1..NWIDTHS                                 ! Range of widths
    RP: range                                           ! Range of cutting patterns
    MAXWIDTH = 94                                       ! Maximum roll width
    EPS = 0.001                                         ! Zero tolerance
    WIDTH: array(WIDTHS) of real                        ! Possible widths
    DEMAND: array(WIDTHS) of integer                    ! Demand per width
    PATTERN: array(WIDTHS, WIDTHS) of integer           ! (Basic) cutting patterns

    use:  array(RP) of mpvar                            ! Rolls per pattern
    soluse:  array(RP) of real                          ! Solution values for variables 'use'
    DemC: array(WIDTHS) of linctr                       ! Demand constraints
    MinRolls:  linctr                                   ! Objective function
    KnapCtr, KnapObj:  linctr                           ! Knapsack constraint & objective
    x:  array(WIDTHS) of mpvar                          ! Knapsack variables
  end-declarations

  getcutstkdata(WIDTH, WDATA, WSIZE)                    ! Get width data
  getcutstkdata(DEMAND, DDATA, DSIZE)                   ! Get demand data

                                                        ! Make basic patterns
  forall(j in WIDTHS) PATTERN(j,j):= floor(MAXWIDTH/WIDTH(j))

  forall(j in WIDTHS) do
    create(use(j))                                      ! Create NWIDTHS variables 'use'
    use(j) is_integer                                   ! Variables are integer and bounded
    use(j) <= integer(ceil(DEMAND(j)/PATTERN(j,j)))
  end-do
  MinRolls:= sum(j in WIDTHS) use(j)                    ! Objective:  minimize no.  of rolls
  forall(i in WIDTHS)                                   ! Satisfy all demands
    DemC(i):= sum(j in WIDTHS) PATTERN(i,j) * use(j) >= DEMAND(i)

  column_gen                                            ! Column generation algorithm
  minimize(MinRolls)                                    ! Solve the resulting problem


!**** Solve the problem z = max{cx :  ax<=b, x in Z^N} with N=NWIDTHS ****
  function knapsack(C:array(range) of real, A:array(range) of real,
                    B:real, xbest:array(range) of integer, pass:integer):  real

!  Hide the demand constraints
    forall(j in WIDTHS) sethidden(DemC(j), true)

!  Define the knapsack problem
    KnapCtr := sum(j in WIDTHS) A(j)*x(j) <= B
    KnapObj := sum(j in WIDTHS) C(j)*x(j)

!  Integrality condition
    if(pass=1) then
      forall(j in WIDTHS) x(j) is_integer
    end-if

    maximize(KnapObj)
    returned:=getobjval
    forall(j in WIDTHS) xbest(j):=round(getsol(x(j)))

!  Reset knapsack constraint and objective, unhide demand constraints
    KnapCtr := 0; KnapObj := 0
    forall(j in WIDTHS) sethidden(DemC(j), false)

  end-function
```

 ©2009 Fair Isaac Corporation. All rights reserved. 

```
!**** Column generation loop at the top node ****
  procedure column_gen
    declarations
      dualdem:  array(WIDTHS) of real
      xbest:  array(WIDTHS) of integer
      ubnd, zbest:  real
      bas:  basis
    end-declarations

    setparam("XPRS_CUTSTRATEGY", 0)              !  Disable automatic cuts
    setparam("XPRS_PRESOLVE", 0)                 !  Switch presolve off
    npatt:=NWIDTHS
    npass:=1

    while(true) do
      minimize(XPRS_LIN, MinRolls)               !  Solve the LP
      savebasis(bas)                             !  Save the current basis
                                                 !  Get the dual values
      forall(i in WIDTHS) dualdem(i):=getdual(DemC(i))
                                                 !  Solve a knapsack problem
      zbest:= knapsack(dualdem, WIDTH, MAXWIDTH, xbest, npass) - 1.0

      if(zbest < EPS) then
        writeln("No profitable column found.\n")
        break
      else
        printpat(zbest, xbest, WIDTH)            !  Print the new pattern
        npatt+=1
        create(use(npatt))                       !  Create a variable for this pattern
        use(npatt) is_integer
        MinRolls+= use(npatt)                    !  Add new var.  to the objective
        ubnd:=0
        forall(i in WIDTHS)
          if(xbest(i) > 0) then
            DemC(i)+= xbest(i)*use(npatt)        !  Add new var.  to demand constraints
            ubnd:= maxlist(ubnd, ceil(DEMAND(i)/xbest(i) ))
          end-if
        use(npatt) <= ubnd                       !  Set upper bound on the new variable

        loadprob(MinRolls)                       !  Reload the problem
        loadbasis(bas)                           !  Load the saved basis
      end-if
    npass+=1
    end-do
  end-procedure

end-model
```

## C program paper.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "xprm_mc.h"
#include "xprm_ni.h"

/* Initialization function of the module 'cutstkdata' */
static int cutstkdata_init(XPRMnifct nifct, int *interver,int *libver, XPRMdsointer **interf);

int main()
{
  XPRMmodel mod;
  int result;
  char params[80];
  static int tabdemand[] = 150, 96, 48, 108, 227;
  static double tabwidth[] =17, 21, 22.5, 24, 29.5;
  int nwidths = 5;

/* Initialize Mosel */
  if(XPRMinit()) return 1;

/* Register 'cutstkdata' as a static module (=stored in the program) */
  if(XPRMregstatdso("cutstkdata", cutstkdata_init)) return 2;

/* Parameters:  the addresses of the data tables and their sizes */
  sprintf(params, "DDATA='%p',DSIZE=%d,WDATA='%p',WSIZE=%d,NWIDTHS=%d",
          tabdemand, sizeof(tabdemand)/sizeof(int), tabwidth,
          sizeof(tabwidth)/sizeof(double), nwidths);

/* Execute the model */
  if(XPRMexecmod("", "paperm.mos", params, &result, &mod)) return 3;

  return result;
}
```


©2009 Fair Isaac Corporation. All rights reserved.

```
/********************** Body of the module 'cutstkdata' ******************/
static int getcutstkdata_int(XPRMcontext ctx,void *libctx);
static int getcutstkdata_dbl(XPRMcontext ctx, void *libctx);
static int printpattern(XPRMcontext ctx, void *libctx);

static XPRMdsofct tabfct[] = {
         {"getcutstkdata", 1000, XPRM_TYP_NOT, 3, "AI.isi", getcutstkdata_int},
         {"getcutstkdata", 1001, XPRM_TYP_NOT, 3, "AI.rsi", getcutstkdata_dbl},
         {"printpat", 1002, XPRM_TYP_NOT, 3, "rAI.iAI.r", printpattern}
};

static XPRMdsointer dsointer = {
         0,NULL,
         sizeof(tabfct)/sizeof(XPRMdsofct),tabfct,
         0,NULL,
         0,NULL
};

static XPRMnifct mm;                     /* To store Mosel's function table */

/**** Initialization function of the module ****/
static int cutstkdata_init(XPRMnifct nifct, int *interver, int *libver, XPRMdsointer **interf)
{
  mm=nifct;                              /* Save the table of functions */
  *interver=XPRM_NIVERS;                 /* The interface version we are using */
  *libver=XPRM_MKVER(0,0,1);             /* The version of the module:  0.0.1 */
  *interf=&dsointer;                     /* Our interface */

  return 0;
}

/**** Initialize an array of integer with data held in C ****/
static int setcutstkdata_int(XPRMcontext ctx,void *libctx)
{
  XPRMarray arr;
  XPRMstring adr_s;
  XPRMset ndxset;
  int *adr, siz, index[1], last, i;

  arr=XPRM_POP_REF(ctx);                 /* The array */
  adr_s=XPRM_POP_STRING(ctx);            /* Data location (as a string) */
  siz=XPRM_POP_INT(ctx);                 /* Data size */
  sscanf(adr_s, "%p", &adr);             /* Get the address from the string */

  mm->getarrsets(arr, &ndxset);
  index[0]=mm->getfirstsetndx(ndxset);
  last=mm->getlastsetndx(ndxset);
  for(i=0;(i<siz) && (index[0]<=last);i++,index[0]++)
    mm->setarrvalint(ctx, arr, index, adr[i]);
  return XPRM_RT_OK;
}

/**** Initialize an array of real with data held in C ****/
static int setcutstkdata_dbl(XPRMcontext ctx,void *libctx)
{
  XPRMarray arr;
  XPRMstring adr_s;
  XPRMset ndxset;
  int siz, index[1], last, i;
  double *adr;

  arr=XPRM_POP_REF(ctx);                 /* The array */
  adr_s=XPRM_POP_STRING(ctx);            /* Data location (as a string) */
  siz=XPRM_POP_INT(ctx);                 /* Data size */
  sscanf(adr_s, "%p", &adr);             /* Get the address from the string */

  mm->getarrsets(arr,&ndxset);
  index[0]=mm->getfirstsetndx(ndxset);
  last=mm->getlastsetndx(ndxset);
  for(i=0;(i<siz) && (index[0]<=last);i++,index[0]++)
    mm->setarrvalreal(ctx, arr, index, adr[i]);
  return XPRM_RT_OK;
}
```


©2009 Fair Isaac Corporation. All rights reserved.

```
/**** Print the new pattern found ****/
static int printpattern(XPRMcontext ctx,void *libctx)
{
  XPRMarray varr, warr;
  int v, index[1];
  double dj, w, tw=0;

  dj=XPRM_POP_REAL(ctx);
  varr=XPRM_POP_REF(ctx);                        /* The value array */
  warr=XPRM_POP_REF(ctx);                        /* The widths array */

  mm->printf(ctx, "New pattern found with marginal cost %g\n", dj);
  mm->printf(ctx, " Widths distribution:  ");
  mm->getfirstarrentry(varr, index);             /* Get the first index tuple */
  do {
    mm->getarrval(varr, index, &v);              /* Get the value */
    mm->getarrval(warr, index, &w);              /* Get the width */
    mm->printf(ctx, "%g:%d ", w, v);
    tw += v*w;
  } while(!mm->getnextarrentry(varr, index));
  mm->printf(ctx, "Total width:  %g\n", tw);
  return XPRM_RT_OK;
}
```

 ©2009 Fair Isaac Corporation. All rights reserved. 