Chapter 5

The basics of Xpress-MP

All problems in this book are formulated using the Xpress-Mosel (for short, Mosel) Language. To run these models, the user has the choice between using the Mosel Command Line Interpreter or the graphical user interface Xpress-IVE. To solve the optimization problems in this book we use the Xpress-Optimizer linear and mixed integer solver by accessing it from the Mosel Language.

All software with reference manuals and the complete set of examples discussed in this book can be downloaded from

http://www.dashoptimization.com/applications_book.html

In the first section of this chapter we show how to execute the small production planning problem from Chapter 1 using the Mosel Command Line Interpreter or Xpress-IVE. Section 5.2 introduces the basics of the Mosel Language.

5.1 Introductory example

A small joinery makes two different sizes of boxwood chess sets. The small set requires 3 hours of machining on a lathe, and the large set requires 2 hours. There are four lathes with skilled operators who each work a 40 hour week, so we have 160 lathe-hours per week. The small chess set requires 1 kg of boxwood, and the large set requires 3 kg. Unfortunately, boxwood is scarce and only 200 kg per week can be obtained. When sold, each of the large chess sets yields a profit of $20, and one of the small chess set has a profit of $5. The problem is to decide how many sets of each kind should be made each week to so as to maximize profit.

In Chapter 1 the transformation of this description into a mathematical model was discussed in some detail and will therefore not be repeated here. An implementation of the model has also already been given (see Section 1.3). We assume that the following Mosel model has been entered into a text file named `chess.mos` (`mos` is the standard file extension expected by Mosel):

```
model Chess
uses "mmxprs" ! We shall use Xpress-Optimizer
declarations
xs, xl: mpvar ! Decision variables: produced quantities
end-declarations
Profit:= 5*xs + 20*xl ! Objective function
Boxwood:= 1*xs + 3*xl <= 200 ! kg of boxwood
Lathe:= 3*xs + 2*xl <= 160 ! Lathehours
maximize(Profit) ! Solve the problem
writeln("LP Solution:") ! Solution printing
```

```
writeln(" Objective: ", getobjval)
writeln("Make ", getsol(xs), " small sets")
writeln("Make ", getsol(xl), " large sets")
end-model
```


5.1.1 Using Xpress-Mosel

Mosel is an advanced modeling and solving language and environment, where optimization problems can be specified and solved with the utmost precision and clarity. The modeling component of Mosel provides an easy to use yet powerful language for describing optimization problems. Through its modular architecture, Mosel provides access to data in different formats (including spreadsheets and databases) and gives access to a variety of solvers, which can find optimal or near-optimal solutions to a problem. Mosel is provided either as a standalone program (the Mosel Command Line Interpreter used in this book) or in the form of libraries that make it possible to embed a model into a larger application written in a programming language.

To run the model we have entered into the file `chess.mos`, we start Mosel at the command prompt, and type the following sequence of commands

```
mosel
exec chess
quit
```

which will start Mosel, compile the model and (if no syntax error has been detected) run the model, and then quit Mosel. We will see output something like that below, where we have highlighted Mosel's output in bold face.

```
mosel
```
**```
** Xpress-Mosel **
(c) Copyright Dash Associates 1998-2006
```**
```
> exec chess
```
**```
LP Solution:
Objective: 1333.33
Make 0 small sets
Make 66.6667 large sets
Returned value: 0
```**
```
> quit
```
**```
Exiting.
```**

The same steps may be done immediately from the command line:

```
mosel -c "exec chess"
```

The `-c` option is followed by a list of commands enclosed in double quotes.
```

If after having started Mosel you type a command that is not recognized by the Mosel Command Line Interpreter (for instance: `h`), Mosel displays the full list of commands (or the possible completions to valid commands) with short explanations.

The different options that may be used from the operating system's command line can be obtained by typing `mosel -h`.

The distribution of Mosel contains several **modules** that add extra functionality to the language. A full list of the functionality of a module can be obtained by using the `exam` command; for instance to see what is provided by the Xpress-Optimizer module `mmxprs`:

```
mosel -c "exam mmxprs"
```

For a complete description of the Mosel Language and the Mosel Command Line Interpreter, the reader is referred to the Mosel Reference Manual, available at

http://www.dashoptimization.com/applications_book.html

From the same address, individual manuals for the Mosel modules can also be downloaded.

5.1.2 Using Xpress-IVE

Xpress-IVE, sometimes called just IVE, is the Xpress **I**nteractive **V**isual **E**nvironment, a complete modeling and optimization development environment running under Microsoft Windows. It presents Mosel in an easy-to-use Graphical User Interface (GUI), with a built-in text editor. IVE can be used for the development, management and execution of multiple models and is ideal for developing and debugging prototype models.

To execute the model file `chess.mos` you need to carry out the following steps.

• Start up IVE.

• Open the model file by choosing *File* > *Open*. The model source is then displayed in the central window (the **IVE Editor**).

• Click the *Run* button (green triangle) or alternatively, choose *Build* > *Run*. The resulting screen display is shown in Figure 5.1.

The **Build** pane at the bottom of the workspace is automatically displayed when compilation starts. If syntax errors are found in the model, they are displayed here, with details of the line and character position where the error was detected and a description of the problem, if available. Clicking on the error takes the user to the offending line.

When a model is run, the **Output/Input** pane at the right hand side of the workspace window is selected to display program output. Any output generated by the model is sent to this window. IVE will also provide graphical representations of how the solution is obtained, which are generated by default whenever a problem is optimized. The right hand window contains a number of panes for this purpose, dependent on the type of problem solved and the particular algorithm used. IVE also allows the user to draw graphs by embedding subroutines in Mosel models (see the documentation on the website for further detail).

IVE makes all information about the solution available through the **Entities** pane in the left hand window. By expanding the list of decision variables in this pane and hovering over one with the mouse pointer, its solution and reduced cost are displayed. Dual and slack values for constraints may also be obtained.

Figure 5.1: Xpress-IVE display after running model `chess.mos`

5.2 Modeling with Mosel

Let us now consider a second, slightly larger model which represents the problem faced by a burglar. With the help of this model we shall explain the basic features of the Mosel language that are used repeatedly in the implementation of the example problems in the following chapters.

5.2.1 The burglar problem

A burglar sees eight items, of different values and weights. He wants to take the items of greatest total value whose total weight is not more than the maximum *WTMAX* he can carry.

We introduce binary variables $take_i$ for all i in the set of all items (*ITEMS*) to represent the decision whether item i is taken or not. $take_i$ has the value 1 if item i is taken and 0 otherwise. Furthermore, let

$VALUE_i$ be the value of item i and $WEIGHT_i$ its weight. A mathematical formulation of the problem is then given by:

maximize

$$\sum_{i \in ITEMS} VALUE_i \cdot take_i \quad \text{(maximize the total value)}$$

$$\sum_{i \in ITEMS} WEIGHT_i \cdot take_i \le WTMAX \quad \text{(weight restriction)}$$

$$\forall i \in ITEMS : take_i \in \{0, 1\}$$

This problem is an example of a **knapsack problem**. It may be implemented with Mosel as follows:

```
model "Burglar 1"
uses "mmxprs"
declarations
ITEMS = 1..8 ! Index range for items
WTMAX = 102 ! Maximum weight allowed
VALUE: array(ITEMS) of real ! Value of items
WEIGHT: array(ITEMS) of real ! Weight of items
take: array(ITEMS) of mpvar ! 1 if we take item i; 0 otherwise
end-declarations
! Item: 1 2 3 4 5 6 7 8
VALUE := [15, 100, 90, 60, 40, 15, 10, 1]
```

```
WEIGHT:= [ 2, 20, 20, 30, 40, 30, 60, 10]
! Objective: maximize total value
MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)
! Weight restriction
sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX
! All variables are 0/1
forall(i in ITEMS) take(i) is_binary
maximize(MaxVal) ! Solve the MIP-problem
! Print out the solution
writeln("Solution:¥n Objective: ", getobjval)
forall(i in ITEMS) writeln(" take(", i, "): ", getsol(take(i)))
end-model
```

When running this model we get the following output:

```
Solution:
Objective: 280
take(1): 1
take(2): 1
take(3): 1
take(4): 1
take(5): 0
take(6): 1
take(7): 0
take(8): 0
```

The **structure** of this model and Mosel models in general is the following:

• **Model:** Every Mosel program starts with the keyword `model`, followed by a name, and terminates with `end-model`.

• **Declarations:** All objects must be declared in a `declarations` block, unless they are defined unambiguously through an assignment (*e.g.* `i:=1` defines `i` as an integer and assigns it the value 1; in our example the objective function `MaxVal` is defined by assigning it a linear expression). There may be several such `declarations` blocks at different places in a model.

• **Problem definition:** Typically, a model starts with the specification of the data (here: assignment of values to `VALUE` and `WEIGHT`), followed by the statement of the problem (here: definition of the objective function `MaxVal`, definition of one inequality constraint, and restricting the variables to be binaries)

• **Solving:** With the procedure `maximize`, we call Xpress-Optimizer to maximize the objective function

`MaxVal`. Since there is no 'default' solver in Mosel, we specify that Xpress-Optimizer is to be used with the statement `uses "mmxprs"` at the beginning of the program.

• **Output printing:** The last two lines print out the value of the optimal solution and the solution values for the decision variables.

• **Line breaks:** It is possible to place several statements on a single line, separating them by semicolons (like `x1 <= 4; x2 >= 7`). Conversely, since there are no special 'line end' or continuation characters, every line of a statement that continues over several lines must end with an operator (`+`, `>=` *etc.*) or characters like `,` that make it obvious that the statement is not terminated.

• **Comments:** As shown in the example, the symbol `!` signifies the start of a comment, which continues to the end of the line. Comments over multiple lines start with `(!` and terminate with `!)`.

We shall now explain certain features used in this model in more detail:

• **Ranges and sets:**

```
ITEMS = 1..8
```

defines a **range set**, that is, a set of consecutive integers from 1 to 8. This range is used as an **index set** for the data arrays (`VALUE` and `WEIGHT`) and for the array of decision variables `take`.

Instead of using numerical indices, we could, for instance, have defined `ITEMS` as a set of strings by replacing the current definition `ITEMS = 1..8` with the following definition:

```
ITEMS = {"camera", "necklace", "vase", "picture", "tv", "video",
"chest", "brick"} ! Index set for items
```

• **Arrays**:

```
VALUE: array(ITEMS) of real
```

defines a one-dimensional array of real values indexed by the range `ITEMS`.

Multi-dimensional arrays are declared in the obvious way *e.g.*

```
VAL3: array(ITEMS, 1..20, ITEMS) of real
```

declares a 3-dimensional real array. Arrays of decision variables (type `mpvar`) are declared likewise, as shown in our example.

All objects (scalars and arrays) declared in Mosel are always initialized with a default value:

`real`, `integer`: 0

`boolean`: `false`

`string`: `''` (*i.e.* the empty string)

The values of data arrays may either be assigned in the model as we show in the example or initialized from file (see Section 5.2.2).

• **Summations:**

```
MaxVal:= sum(i in Items) VALUE(i)*x(i)
```

defines a linear expression called `MaxVal` as the sum

X

i2Items

$VALUE_i \cdot x_i$

• **Simple looping:**

`forall(i in ITEMS) take(i) is_binary`

illustrates looping over all values in an index range. Recall that the index range `ITEMS` is 1, ..., 8, so the statement says that `take(1)`, `take(2)`, ..., `take(8)` are all binary variables.

There is another example of the use of `forall` at the penultimate line of the model when writing out all the solution values.

Other types of loops are used in some of the application examples (see the classification tables at the beginning of Part II).

• **Integer Programming variable types:**

To make an `mpvar` variable, say variable `xbinvar`, into a binary (0/1) variable, we just have to say

`xbinvar is_binary`

To make an `mpvar` variable an integer variable, *i.e.* one that can only take on integral values in a MIP problem, we would have

`xintvar is_integer`

5.2.2 Reading data from text files

The following example illustrates how data may be read into tables from text files. In the Burglar problem instead of having the item data embedded in the model file we have the data in a file. We might have the following Mosel model in a file `burglar2.mos`.

```
model "Burglar 2"
uses "mmxprs"
declarations
ITEMS: set of string ! Set of items
WTMAX = 102 ! Maximum weight allowed
VALUE: array(ITEMS) of real ! Value of items
WEIGHT: array(ITEMS) of real ! Weight of items
end-declarations
initializations from 'burglar.dat'
VALUE
WEIGHT
end-initializations
declarations
take: array(ITEMS) of mpvar ! 1 if we take item i; 0 otherwise
```

```
end-declarations
! Objective: maximize total value
MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)
! Weight restriction
sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX
! All variables are 0/1
forall(i in ITEMS) take(i) is_binary
maximize(MaxVal) ! Solve the MIP-problem
! Print out the solution
writeln("Solution:¥n Objective: ", getobjval)
forall(i in ITEMS) writeln(" take(", i, "): ", getsol(take(i)))
end-model
```

The file `burglar.dat` contains

```
VALUE: [("camera") 15 ("necklace") 100 ("vase") 90 ("picture") 60
("tv") 40 ("video") 15 ("chest") 10 ("brick") 1]
WEIGHT:[("camera") 2 ("necklace") 20 ("vase") 20 ("picture") 30
("tv") 40 ("video") 30 ("chest") 60 ("brick") 10]
```

The `initializations` block tells Mosel where to get data from to initialize sets and arrays. The order of the data items in the file does not have to be the same as that in the `initializations` block. Note that the contents of the set `ITEMS` is defined indirectly through the index values of the arrays `VALUE` and `WEIGHT`. We only declare the variables once the data has been initialized and hence, the set `ITEMS` is known.

In the application examples, where appropriate, we show how to work with **dynamic** arrays of data and decision variables (see the classification tables at the beginning of Part II).

The data may also be given in the form of a single record, say, `KNAPSACK`. The initialization then takes the following form:

```
initializations from 'burglar2.dat'
[VALUE, WEIGHT] as 'KNAPSACK'
end-initializations
```

and the data file `burglar2.dat` has the following contents:

```
KNAPSACK: [ ("camera") [ 15 2]
("necklace") [100 20]
```



```
("vase") [ 90 20]
("picture") [ 60 30]
("tv") [ 40 40]
```

```
("video") [ 15 30]
("chest") [ 10 60]
("brick") [ 1 10] ]
```

In the examples of this book we always read data from text files. However, with Mosel it is also possible to read and write data from/to other sources (such as spreadsheets and databases) or input data in memory. For further information, the reader is referred to the documentation on the website.

5.2.3 Reserved words

The following words are reserved in Mosel. The upper case versions are also reserved (*i.e.* `AND` and `and` are keywords but not `And`). Do not use them in a model except with their built-in meaning.

`and`, `array`, `as`

`boolean`, `break`

`case`

`declarations`, `div`, `do`, `dynamic`

`elif`, `else`, `end`

`false`, `forall`, `forward`, `from`, `function`

`if`, `in`, `include`, `initialisations`, `initializations`, `integer`, `inter`,

`is_binary`, `is_continuous`, `is_free`, `is_integer`, `is_partint`, `is_semcont`,

`is_semint`, `is_sos1`, `is_sos2`

`linctr`

`max`, `min`, `mod`, `model`, `mpvar`

`next`, `not`

`of`, `options`, `or`

`parameters`, `procedure`, `public`, `prod`

`range`, `real`, `repeat`

`set`, `string`, `sum`

`then`, `to`, `true`

`union`, `until`, `uses`

`while`