Introduction to OPL

Fabio Martignon

Sources:

https://www.ibm.com/docs/en/SSSA5P_12.8.0/ilog.odms.studio.help/pdf/opl_languser.pdf

https://perso.ensta-paris.fr/~diam/ro/online/cplex/cplex1271/OPL_Studio/opllanguser/topics/opl_languser_intro.html

Introduction

- OPL is a modeling language
- It can be used, as we will do, to model several networking scenarios, for example:
  - 5G Networks (Multi Connectivity etc…)
  - Network planning
  - Routing with different constraints
  - …

Linear programming

- **The production planning problem**
- It describes a linear programming problem.
- Consider a Belgian company, Volsay, which specializes in producing ammoniac gas (NH3) and ammonium chloride (NH4Cl).
- Volsay has at its disposal
  - 50 units of nitrogen (N),
  - 180 units of hydrogen (H), and
  - 40 units of chlorine (Cl).
- The company makes a profit of 40 Euros for each sale of an ammoniac gas unit and 50 Euros for each sale of an ammonium chloride unit.
- Volsay would like a production plan maximizing its profits given its available stocks.
- The following OPL statement formalizes this problem.

Linear programming

- The following OPL statement formalizes this problem.

- A simple Production Problem (volsay.mod)

```
dvar float+ Gas;
dvar float+ Chloride;

maximize   40 * Gas + 50 * Chloride;

subject to {
        ctMaxTotal:
                Gas + Chloride <= 50;
        ctMaxTotal2:
                 3 * Gas + 4 * Chloride <= 180;
         ctMaxChloride:
                Chloride <= 40;
           }
```

This statement declares two real decision variables, gas and chloride, representing the production of ammoniac gas and ammonium chloride. These variables are of type *float*.

Linear programming

- This statement declares two real decision variables, gas and chloride, representing the production of ammoniac gas and ammonium chloride. These variables are of type float.

```
dvar float+ Gas;
dvar float+ Chloride;
```

- The ***objective function***

```
maximize   40 * Gas + 50 * Chloride;
```

states that the profit must be *maximized*.

- The ***constraints*** ensure that the production plan does not exceed the available stocks of *nitrogen*, *hydrogen*, and *chlorine*, respectively.
  - The constraint gas + chloride <= 50 represents the capacity constraint for *nitrogen*, since each unit of ammoniac gas and of ammonium chloride uses one unit of nitrogen.
  - The next two constraints, for hydrogen and chlorine respectively, are similar in nature.
- A solution to an optimization problem is typically an assignment of values to the variables that satisfies the constraints and optimizes the objective function.

- **Note** that in A simple production problem (volsay.mod), the constraints are identified with so-called "*labels*". It is recommended to label constraints in a model.

Linear programming

- **A solution to volsay.mod**
- For the Volsay production-planning problem, OPL returns the optimal solution

```
Final Solution with objective 2300.0000:
gas = 20.0000;
chloride = 30.0000;
```

Elements od the Production Model

- We now describe the details of this Linear Programming (LP) model.
- The Volsay model shown above is a ***linear programming*** model.
- Linear programming is the class of problems that can be expressed as the optimization of a *linear objective* function subject to *a set of linear constraints* (i.e., linear equations and inequalities) over **real numbers**.
- Linear programming models can be solved for large numbers of variables and constraints and are, from a computational standpoint, the simplest applications we will consider.

Elements od the Production Model

- In the following we will examine:
  - Arrays
  - Data declarations
  - Aggregate operators and quantifiers
  - Isolating the data
  - Data initialization
  - Tuples
  - Displaying results
  - Setting CPLEX parameters
  - Integer programming: the knapsack problem
  - Mixed integer-linear programming: a blending problem

Arrays

- The previous model formulation is very specific to the application at hand.
- In general, it is desirable to write *generic models* that can be extended, modified easily, and applied in different contexts.
- We will now describe a number of OPL concepts to simplify the process of creating such models.
- A first step towards more genericity is the use of **arrays**, which makes it easier, for instance, to accommodate new products in the future.

Arrays

- The Volsay production planning model can be rewritten using arrays as:

The volsay production model with arrays

```
{string} Products =  {"gas","chloride"};
dvar float production[Products];
maximize
      40 * production["gas"] + 50 * production["chloride"];
subject to {
     production["gas"] + production["chloride"] <= 50;
     3 * production["gas"] + 4 * production["chloride"] <= 180;
     production["chloride"] <= 40;
}
```

Arrays

- This new statement illustrates several features of the language. First, the instruction

```
{string} Products =  {"gas","chloride"};
```

declares a set of strings *Products* that represents the set of products of the company. The declaration

-
```
dvar float production[Products];
```

declares an array of 2 decision variables, production["gas”] and production["chloride"], to represent the optimal production of ammoniac gas and ammonium chloride.

These decision variables are used in the rest of the statement, which remains essentially the same as the previous model.

One of the key features of OPL is the generality of its arrays: OPL arrays can have an arbitrary number of dimensions and their index sets can be arbitrary finite sets, possibly involving complex data structures.

Data Declarations

- A second fundamental step towards more genericity in the model amounts to representing **the problem data** explicitly.
- In addition to the products, the problem data obviously consists of the (1) *components* (nitrogen, hydrogen, and chloride), the (2) *demand* of each product for each component, the (3) *profit* of each product, and the (4) *stock* available for each component.
- The following example, **gas.dat,** declares and initializes the data.
- **Declaring and initializing data (gas.dat)**

```
Products = { "gas" "chloride" }; Components = { "nitrogen" "hydrogen" "chlorine" };  Demand = [ [1 3 0] [1 4 1] ]; Profit = [30 40]; Stock = [50 180 40];
```

Data Declarations

- The data element *Components* is a set of strings that defines the chemical components necessary for the products,
- *Demand* is a two-dimensional array whose element *Demand[p][c]* represents the demand of product *p* for component *c*, and
- *Profit* and *Stock* are two arrays representing the profit of each product and the stock available for each component.
- The rest of the statement can be obtained easily by replacing the numbers by the relevant data items.

For instance, the objective function is simply written as

```
maximize
      sum( p in Products )
            Profit[p] * Production[p];
```

Aggregate Operators and Quantifiers

- The previous statement contains much redundancy. All constraints, and all arithmetic terms in these constraints and in the objective function, are similar: they differ only in their *indices*.
- OPL has two features to factorize these commonalities, aggregate operators and quantifiers, as shown in the following model, gas1.mod.

A simple production model (gas1.mod).

```
{string} Products = { "gas", "chloride" };
{string} Components = { "nitrogen", "hydrogen", "chlorine" };

float Demand[Products][Components] = [ [1, 3, 0], [1, 4, 1] ];
float Profit[Products] = [30, 40];
float Stock[Components] = [50, 180, 40];

dvar float+ Production[Products];

maximize
        sum( p in Products )
                Profit[p] * Production[p];
subject to {
        forall( c in Components )
         ct:
                sum( p in Products )
                        Demand[p][c] * Production[p] <= Stock[c];
 }
```

Aggregate Operators and Quantifiers

- The **objective function**

```
maximize
        sum( p in Products )
                Profit[p] * Production[p];
```

illustrates the use of the aggregate operator sum to take the summation of the individual profits. A variety of aggregate operators are available in OPL, including ***sum***, ***prod***, ***min***, and ***max***.

- The instruction

```
subject to {
        forall( c in Components )
                ct:
                        sum( p in Products )
                                Demand[p][c] * Production[p] <= Stock[c];
}
```

shows how the universal quantifier ***forall*** can be used to state closely related constraints.

It generates one constraint for each chemical component, each constraint stating that the total demand for the component cannot exceed its available stock. OPL supports rich parameter specifications in aggregate operators and quantifiers (see Expressions in the *Language Reference Manual*).

Isolating the Data

- Another fundamental step in making models reusable is to separate the model and the *instance data*. OPL supports this clean separation through the notion of ***projects***.
- A project is the association of a model file, one or more data files (optional), and one or more settings files (optional), associated in run configurations. A minimal project has one run configuration containing only one model.
- Model files use the file name extension *.mod* while data files use the file name extension *.dat*. The model declares the data but does not initialize it. The data files contain the initialization instructions for each declared data item.

(See Understanding OPL projects in *Quick Start*.)

- Here we do not describe the details of IBM ILOG OPL, but generally describe applications by giving the model and the instance data separately

Isolating the Data

- For instance, the following model, *gas.mod*, and the instance data, *gas.dat*, together make up a project for the Volsay production-planning problem. The model part is essentially the same as the one presented earlier, except that it declares the data but does not initialize it.
- **The production model (gas.mod)**

```
{string} Products = …;
{string} Components = …;

float Demand[Products][Components] = …;
float Profit[Products] = …;
float Stock[Components] = …;

dvar float+ Production[Products];

maximize
        sum( p in Products )
                Profit[p] * Production[p];
subject to {
        forall( c in Components )
         ct:
                sum( p in Products )
                        Demand[p][c] * Production[p] <= Stock[c];
 }
```

Isolating the Data

- A declaration of the form

```
float profit[Products] = ...;
```

declares the array *profit* and specifies that its initialization is given in a data file. The data file simply associates an initialization with each non-initialized piece of data.

- **Instance data for the production model (gas.dat)**

```
Products = { "gas" "chloride" };
Components = { "nitrogen" "hydrogen" "chlorine" };

Demand = [ [1 3 0] [1 4 1] ];
Profit = [30 40];
Stock = [50 180 40];
```

Data Initialization

- OPL offers a variety of ways of initializing data. One particularly useful feature is the possibility of associating indices with values to avoid various kinds of errors. The following instance data, *gasn.dat*, illustrates this feature on the instance data for the Volsay production model.

- **Instance data with indices for the production model (gasn.dat)**

```
Products = { "gas", "chloride" };
Components = { "nitrogen", "hydrogen", "chlorine" };

Profit = #["gas":30, "chloride":40]#;
Stock = #["nitrogen":50, "hydrogen":180, "chlorine":40]#;
Demand = #[
                    "gas":     #[ "nitrogen":1 "hydrogen":3 "chlorine":0 ]#,
                    "chloride": #[ "nitrogen":1 "hydrogen":4 "chlorine":1 ]#
                ]#;
```

Data Initialization

- The initialization

```
profit = #["gas":30 "chloride":40]#;
```

- describes the initialization of array *profit* by associating the value 30 with index *gas* and the value 40 with index *chloride*. (Of course, the *order* of the pairs has *no importance* in these initializations.)
- When using *index:value* pairs, the delimiters #[and]# must be used instead of [and].
- Note also that, in data files, the items can be initialized in any order and the commas can be omitted freely.

Tuples

- OPL offers a variety of data structures in addition to arrays and sets of strings. **Tuples**, a fundamental tool for structuring the application data, offer an alternative to the traditional approach of representing data in parallel arrays.
- To see the use of tuples in OPL, consider the following production-planning model. To meet the demands of its customers, a company manufactures its products in its own factories (*inside* production) or buys them from other companies (*outside* production).
- Inside production is subject to some *resource constraints*: each product consumes a certain amount of each resource. In contrast, outside production is theoretically unlimited.

- The problem is to determine how much of each product should be produced inside and outside the company while minimizing the overall production cost, meeting the demand, and satisfying the resource constraints.

- The following example, *production.mod*, depicts an OPL model for this problem that uses only the concepts introduced so far, and *production.dat* presents the data for a specific instance.

Tuples

- **A production-planning problem (production.mod)**

```
{string} Products = ...;
{string} Resources = ...;

float Consumption[Products][Resources] = ...;
float Capacity[Resources] = ...;
float Demand[Products] = ...;
float InsideCost[Products] = ...;
float OutsideCost[Products]  = ...;

dvar float+ Inside[Products];
dvar float+ Outside[Products];

minimize
        sum( p in Products )
         ( InsideCost[p] * Inside[p] + OutsideCost[p] * Outside[p] );

subject to {
        forall( r in Resources )
            ctCapacity:
                sum( p in Products )
                 Consumption[p][r] * Inside[p] <= Capacity[r];

        forall(p in Products)
          ctDemand:
                Inside[p] + Outside[p] >= Demand[p];
 }
```

Tuples

- An instance of the problem must specify the products, the resources, the capacity of the resources, the demand for each product, the consumption of resources by the different products, and the inside and outside costs of each product.
- These various data items are specified in the standard way in *production.dat* below. The model contains two arrays of variables: element *Inside[p]* (respectively *Outside[p]*) represents the inside (respectively outside) production of product *p*. The objective function specifies that the production cost must be minimized.

Data for the production-planning problem (production.dat)

```
Products =  { "kluski", "capellini", "fettuccine" };
Resources = { "flour", "eggs" };

Consumption = [ [0.5, 0.2], [0.4, 0.4], [0.3, 0.6] ];
Capacity = [ 20, 40 ];
Demand = [ 100, 200, 300 ];
InsideCost = [ 0.6, 0.8, 0.3 ];
OutsideCost  = [ 0.8, 0.9, 0.4 ];
```

Tuples

- The production cost is simply the sum of the individual production costs, which are obtained by multiplying the inside and outside productions of the given product by their respective costs.
- Finally, the model has two types of constraints. The first set of constraints expresses the *capacity constraints*, the second set states the *demand constraints*.
- The model is once again a *linear programming problem*.

A solution to production.mod

- For the instance data given in Data for the production-planning problem (production.dat), OPL outputs the following solution:

```
Final Solution with objective 372.0000:
      inside = [40.0000 0.0000 0.0000];
      outside = [60.0000 200.0000 300.0000];
```

- Although the model is simple, it is inconvenient in separating the data associated with each product in different arrays: for instance, array demand stores the demand for the products, while array insideCost stores their inside costs.
- This technique, sometimes called *parallel arrays*, may be error-prone and less readable for more complicated models.
- Tuples provide a simple way to cluster related data and impose more structure on a model.
- This is illustrated in the revisited example below, *product.mod*, and the revised data *product.dat*, which exhibit an alternative model for the production-planning problem.

A solution to production.mod

- **The production-planning problem revisited (product.mod)**

```
{string} Products = ...;
{string} Resources = ...;
tuple productData {
        float demand;
        float insideCost;
        float outsideCost;
        float consumption[Resources];
}
productData Product[Products] = ...;
float Capacity[Resources] = ...;

dvar float+ Inside[Products];
dvar float+ Outside[Products];

execute CPX_PARAM {
        cplex.preind = 0;
         cplex.simdisplay = 2;
}
```

A solution to production.mod

- **The production-planning problem revisited (product.mod)**

```
minimize
      sum( p in Products )
       (Product[p].insideCost * Inside[p] +      Product[p].outsideCost *
Outside[p] );
subject to {
      forall( r in Resources )
      ctInside:
            sum( p in Products )
                  Product[p].consumption[r] * Inside[p] <= Capacity[r];
      forall( p in Products )
            ctDemand:
                  Inside[p] + Outside[p] >= Product[p].demand;
}
```

Data for the revised production-planning problem (*product.dat*)

```
Products =  { "kluski", "capellini", "fettuccine" };
Resources = { "flour", "eggs" };
Product = #[
    kluski : < 100, 0.6, 0.8, [ 0.5, 0.2 ] >,
    capellini : < 200, 0.8, 0.9, [ 0.4, 0.4 ] >,
    fettuccine : < 300, 0.3, 0.4, [ 0.3, 0.6 ] >
            ]#;
Capacity = [ 20, 40 ];
```

Data for the revised production-planning problem (*product.dat*)

- The instruction

```
tuple productData {
float demand;
float insideCost;
float outsideCost;
float consumption[Resources];
}
```

- declares a tuple type with four fields. The first three fields, of type *float*, are used to represent the demand and costs of a product; the last field is an array representing the resource consumptions of the product. These fields are intended to hold all the data related to a given product.

Data for the revised production-planning problem (*product.dat*)

- The instruction

```
ProductData product[Products] = ...;
```

- declares an array of these tuples, one for each product.

- The initialization …

```
Product = #[
    kluski : < 100, 0.6, 0.8, [ 0.5, 0.2 ] >,
    capellini : < 200, 0.8, 0.9, [ 0.4, 0.4 ] >,
    fettuccine : < 300, 0.3, 0.4, [ 0.3, 0.6 ] >
            ]#;
```

Data for the revised production-planning problem (*product.dat*)

- … from "Data for the revised production-planning problem" (*product.dat*) specifies these various data items: tuples are initialized by giving values for each of their fields. It is of course possible to use a named initialization for the tuple, as shown in "Named data for the revised production-planning problem" (*productn.dat*), in which case the initialization is enclosed with #< and >#.
- Tuple fields can be obtained by suffixing the tuple with a dot and the field name. For instance, in the objective function

```
minimize
      sum( p in Products )
            (Product[p].insideCost * Inside[p] +
           Product[p].outsideCost * Outside[p] );
```

the expression *product[p].insideCost* represents the field *insideCost* of the tuple *product[p]*.

Data for the revised production-planning problem (*product.dat*)

- Similarly, in the constraint

```
forall(r in Resources)
    sum(p in Products) product[p].consumption[r] * inside[p] <= capacity[r];
```

the expression *product[p].consumption* represents the field consumption of tuple *product[p]*.

This field is an array that can be subscripted in the traditional way.

Displaying the results

- We now describe how to display results by writing an execute IBM ILOG Script block.
- The statements presented so far did not specify what elements of the solution should be displayed. OPL offers a way to display the results of an application. An interesting feature of OPL is the ability to display *tuples of expressions*.
- **Procedure**
- To display results using an execute block:

  1) Add the following IBM ILOG Script execute block to the product.mod file (see The production-planning problem revisited (product.mod))

```
tuple R { float x; float y; };
{R} Result = { <Inside[p],Outside[p]> | p in Products };
execute { writeln("Result=",Result); }
```

Displaying the results

You see the following output:

```
Optimal solution found with objective: 372
result= {<40.0000 60.0000> <0.0000 200.0000> <0.0000 300.0000>}
```

- Run the product model with the productn.dat data file shown in Named data for the revised production-planning problem (productn.dat).
- You can visualize the inside and outside productions of a product simultaneously.

```
Final Solution with objective 372.0000:
    inside = [40.0000 0.0000 0.0000];
    outside = [60.0000 200.0000 300.0000];
```

Displaying the results

- **Named data for the revised production-planning problem (productn.dat)**

```
Products =  { "kluski", "capellini", "fettuccine" };
Resources = { "flour", "eggs" };

Product = #[
                    kluski :
          #< demand:100
                          insideCost:0.6
                          outsideCost:0.8
                          consumption:[0.5 0.2]
                           >#,
                   capellini :
          #< demand:200
                        insideCost:0.8
                        outsideCost:0.9
                        consumption:[0.4 0.4]   >#,
               fettuccine :
  #< demand:300
                   insideCost:0.3
                    outsideCost:0.4
                   consumption:[0.3 0.6]
 >#
     ]#;

Capacity = [ 20, 40 ];
```

Displaying the results

- Add the following IBM ILOG Script postprocessing lines to the *product.mod* file

```
execute {
      for(p in Products)
      writeln("inside[",p,"].reducedCost = ", inside[p].reducedCost);
}
```

- You can see both the inside production of a product and its reduced cost.

```
Optimal solution found with objective: 372
inside[kluski].reducedCost = 0
inside[capellini].reducedCost = 0.06000000000000005
inside[fettuccine].reducedCost = 0.02000000000000002
```

Integer Programming

- Integer programming expresses the optimization of a **linear function** subject to a set of linear constraints over **integer variables**.
- The statements presented in Linear programming: a production planning example are all linear programming models. However, linear programs with very large numbers of variables and constraints can be solved efficiently.
- Unfortunately, this is no longer true when the variables are required to take integer values.
- *Integer programming* is the class of problems that can be expressed as the **optimization of a linear function subject to a set of linear constraints over integer variables**.

Integer Programming

- It is in fact **NP-hard**.
- More important, perhaps, is the fact that the integer programs that can be solved to provable optimality in reasonable time are much smaller in size than their linear programming counterparts.
- There are exceptions, of course, and several important classes of integer programs can still be solved efficiently, but users of OPL should be warned that discrete problems are in general **much harder to solve than linear programs**.

The Knapsack problem

- A typical example of integer program is **the knapsack problem**, which can be intuitively understood as follows.
- We have a knapsack with a **fixed capacity** (an integer) and a **number of items**. Each item has an associated **weight** (an integer) and an associated **value** (another integer).
- The problem consists of filling the knapsack without exceeding its capacity, while maximizing the overall value of its contents.
- A **multi-knapsack problem** is similar to the knapsack problem, except that there are *multiple features* for the object (e.g., weight and volume) and multiple capacity constraints.

- The following example, knapsack.mod, depicts a model for the multi-knapsack problem, while Data for the multi-knapsack problem (knapsack.dat) describes an instance of the problem

The Knapsack problem

A multi-knapsack model (knapsack.mod)

```
int NbItems = ...;
int NbResources = ...;
range Items = 1..NbItems;
range Resources = 1..NbResources;
int Capacity[Resources] = ...;
int Value[Items] = ...;
int Use[Resources][Items] = ...;
int MaxValue = max(r in Resources) Capacity[r];

dvar int Take[Items] in 0..MaxValue;

maximize
        sum(i in Items) Value[i] * Take[i];

subject to {
         forall( r in Resources )
                  ct:
                           sum( i in Items )
                                    Use[r][i] * Take[i] <= Capacity[r];
}
```

The Knapsack problem

- This model has several novel features. It represents items and resources not by string sets but rather by **integers**. In other words, the items (respectively the resources) are represented by **successive integers starting at 1**.
- The instructions

```
int NbItems = ...;
int NbResources = ...;
range Items = 1..NbItems;
range Resources = 1..NbResources;
```

declare the number of items and the number of resources, as well as two ranges, Items and Resources, to represent the set of items and the set of resources.

The Knapsack problem

- The next three instructions

```
int Capacity[Resources] = ...;
int Value[Items] = ...;
int Use[Resources][Items] = ...;
```

are similar to the data declarations presented in Data declarations and the subsequent sections.

The array *Capacity* represents the capacity of the resources, the array *Value* the value of each item, and *Use[r][i]* the use of resource *r* by item *i*.

The Knapsack problem

- The next instruction

```
int MaxValue = max(r in Resources) Capacity[r];
```

is more interesting.

- It declares an integer MaxValue whose value is given by an expression.
- OPL and IBM ILOG Script have many features for computing and preprocessing data, since this is fundamental in simplifying and improving the efficiency of many models.

The Knapsack problem

- The instruction

```
dvar int Take[Items] in 0..MaxValue;
```

declares the problem variables: *take[Items]* represents the number of times item i is selected in the solution. The variable is of type integer and is restricted to range in *0..MaxValue*.

- The rest of the statement is rather standard and should raise no difficulty.
- Data for the multi-knapsack problem (knapsack.dat) describes an instance of the problem.

The Knapsack problem - Dat

- **Data for the multi-knapsack problem (knapsack.dat)**

```
NbResources = 7;
NbItems = 12;
Capacity = [ 18209, 7692, 1333, 924, 26638, 61188, 13360 ];
Value = [ 96, 76, 56, 11, 86, 10, 66, 86, 83, 12, 9, 81 ];
Use = [
        [ 19,    1,  10,   1,    1,   14, 152, 11,  1,    1,  1,  1 ],
        [  0,    4,  53,   0,    0,   80,    0,  4,  5,    0,  0,  0 ],
        [  4, 660,   3,   0,   30,    0,    3,  0,  4,  90,  0,  0],
        [  7,    0,  18,   6, 770, 330,    7,  0,  0,    6,  0,  0],
        [  0,   20,   0,   4,   52,    3,    0,  0,  0,    5,  4,  0],
        [  0,    0,  40,  70,    4,   63,    0,  0, 60,    0,  4,  0],
        [  0,   32,   0,   0,    0,    5,    0,  3,  0, 660,  0,  9]];
```

The Knapsack problem - Solution

- For the instance of the problem specified in Data for the multi-knapsack problem (knapsack.dat), here are the final solution and the solutions that satisfy all the constraints but are not the best with respect to the objective function:

Feasible solution with objective 261890.0000:
take = [0 0 0 154 0 0 0 912 333 0 6505 1180];

Feasible solution with objective 261916.0000:
take = [0 0 0 153 0 0 0 912 333 0 6499 1180];

Final solution with objective 261916.0000:
take = [0 0 0 154 0 0 0 913 333 0 6499 1180];

The Knapsack problem - Solution

- Although integer programs are, in general, substantially harder to solve than linear programs, they have also been the topic of intensive investigation. OPL recognizes when a statement is an integer programming model and uses CPLEX algorithms to solve it.
- **Note:** The results of objectives may vary, according to the machine and the version of CPLEX used.

Mixed integer-linear programs

- **Mixed integer-linear Programs (MILP)** include both *integer* and *real* variables.
- OPL can also solve models that include both integer and real variables, generally known as mixed integer-linear programs (MILP).
- OPL approaches them in essentially the same way as integer programs. A **branch-and-bound** algorithm can exploit the linear relaxation except, of course, that branching takes place only on integer variables.

The blending problem

- The following **blending problem** is taken from W. Winston's book (see the Bibliography).
- Consider the following application involving mixing some metals into an alloy. The metal may come from several sources: in pure form or from raw materials, scraps from previous mixes, or ingots.
- The alloy must contain a certain amount of the various metals, as expressed by a production constraint specifying lower and upper bounds for the quantity of each metal in the alloy. Each source also has a cost and the problem consists of blending the sources into the alloy while minimizing the cost and satisfying the production constraints.
- Similar problems arise in other domains, e.g., the oil, paint, and the food processing industries. The following example shows the two parts of the model for the blending problem.

The blending problem

- Blending problem: blending.mod file

```
int   NbMetals = ...;
int   NbRaw = ...;
int   NbScrap = ...;
int   NbIngo = ...;

range Metals = 1..NbMetals;
range Raws = 1..NbRaw;
range Scraps = 1..NbScrap;
range Ingos = 1..NbIngo;

float CostMetal[Metals] = ...;
float CostRaw[Raws] = ...;
float CostScrap[Scraps] = ...;
float CostIngo[Ingos] = ...;
float Low[Metals] = ...;
float Up[Metals] = ...;
float PercRaw[Metals][Raws] = ...;
float PercScrap[Metals][Scraps] = ...;
float PercIngo[Metals][Ingos] = ...;

int Alloy  = ...;
```

The blending problem

- Blending problem: blending.mod file

```
dvar float+   p[Metals];
dvar float+   r[Raws];
dvar float+   s[Scraps];
dvar int+     i[Ingos];
dvar float    m[j in Metals] in Low[j] * Alloy .. Up[j] * Alloy;


minimize
 sum(j in Metals) CostMetal[j] * p[j] +
 sum(j in Raws)   CostRaw[j]   * r[j] +
 sum(j in Scraps) CostScrap[j] * s[j] +
 sum(j in Ingos)  CostIngo[j]  * i[j];
subject to {
 forall( j in Metals )
   ct1:
     m[j] ==
     p[j] +
     sum( k in Raws )   PercRaw[j][k] * r[k] +
     sum( k in Scraps ) PercScrap[j][k] * s[k] +
     sum( k in Ingos )  PercIngo[j][k] * i[k];
   ct2:
     sum( j in Metals ) m[j] == Alloy;
}
```

Elements of the Blending problem

- **Problem data**
- The model is described in terms of a number of constants specifying the various types of metals, raw materials, scrap, and ingots.
- In the instance data shown in Instance data for the blending problem (blending.dat), there are three metals, two raw materials, two kinds of scrap, and one kind of ingot.
- The model also defines ranges for each of the components. It then defines the cost of the various components in costMetal, costRaw, costScrap, costIngo. In the instance data, for example, the second raw material has a cost of 5.
- The data items Low and Up specify the production constraints and give lower and upper bounds on the quantity of each sort of metal in the alloy. For example, in the instance data, between 30% and 40% of the alloy must be the second metal.
- The next data items, percRaw, percScrap, and percIngo, specify the percentage of each metal in the sources. In Instance data for the blending problem (blending.dat), the second type of scrap contains 1% of the first metal, none of the second metal, and 70% of the third metal.
- Finally, the data alloy specifies the amount of alloy to be produced.

The blending problem – Dat file

- Blending problem: blending.dat file

```
NbMetals = 3;
NbRaw = 2;
NbScrap = 2;
NbIngo = 1;

CostMetal = [22, 10, 13];
CostRaw = [6, 5];
CostScrap = [ 7, 8];
CostIngo = [ 9 ];
Low = [0.05, 0.30, 0.60];
Up = [0.10, 0.40, 0.80];
PercRaw = [ [ 0.20, 0.01 ], [ 0.05, 0 ], [ 0.05, 0.30 ] ];
PercScrap = [ [ 0 , 0.01 ], [ 0.60, 0 ], [ 0.40, 0.70 ] ];
PercIngo = [ [ 0.10 ], [ 0.45 ], [ 0.45 ] ];
Alloy  = 71;
```

The blending problem – Decision Variables

- Decision variables

The decision variables specify how much of each source is used in the alloy: the array *p* specifies the quantities of pure metals, array *r* specifies the quantities of raw materials, array *s* specifies the quantities of scrap, array *i* specifies the number of ingots. All variables are of type **float** except number of ingots, which are **integers**.

The problem is thus a **mixed integer-linear program**. The instruction

```
dvar float m[j in Metals] in low[j] * alloy .. up[j] * alloy;
```

is particularly interesting, since it shows how to specify the range of decision variables in a generic fashion. More precisely, the range of variables *m[j]* is given by the expression:

```
low[j] * alloy .. up[j] * alloy
```

Note also that the model uses the variables in array m as intermediary variables to represent the quantity of each metal produced.

The blending problem - Constraints

There are two types of constraints in this problem.

The *forall* constraint

```
subject to {
  forall( j in Metals )
    ct1:
      m[j] ==
      p[j] +
      sum( k in Raws )   PercRaw[j][k] * r[k] +
      sum( k in Scraps ) PercScrap[j][k] * s[k] +
      sum( k in Ingos )  PercIngo[j][k] * i[k];
    ct2:
      sum( j in Metals ) m[j] == Alloy;
}
```

makes sure that the right amounts of metal are produced. The amount *m[j]* of metal *j* must be equal to the amount of pure metal p[j] added to the quantity of metal *j* contained in the raw materials, the scrap, and the ingots. The correct amount of metals are computed using the percentage of metals contained in the sources.

The *sum* constraint

```
sum(j in Metals) m[j] == alloy;
```

makes sure that the various metals produced give the correct amount of alloy. The objective function in this model is rather simple. It consists of computing the price of each source from its unit price (e.g., costMetal) and the amount produced (e.g., *p[j]*).

A solution for the blending problem

For the instance data given in Instance data for the blending problem (blending.dat), OPL returns the solution

Final Solution with objective 653.6100:

p = [0.0467 0.0000 0.0000];

r = [0.0000 0.0000];

s = [17.4167 30.3333];

i = [32];

m = [3.5500 24.8500 42.6000];