

KNAPSACK PROBLEM

1. PROBLEM DESCRIPTION

The (smuggler's) knapsack is a well known Integer programming problem. Here we consider the simple variant and the multi-knapsack problem.

1.1. Simple variant. Given n objects, each with an associated volume and value, and a knapsack of bounded volume v . The objective is to put as many objects into the knapsack such that the overall value is maximised.¹ An Integer programming formulation of the problem can be found e.g. in [1, p. 44]: the n objects have associated volumes $a_1 \dots a_n$ and values $b_1 \dots b_n$, the knapsack volume being v . There are further n variables $x_1 \dots x_n$, all having the domain $\{0, 1\}$. Variable x_i is 1 (0) if object i is (not) contained in the knapsack. The constraints are thus expressed as:

$$\sum_{i=1}^n a_i * x_i \leq v$$

and the objective function is expressed as

$$\sum_{i=1}^n b_i * x_i$$

1.2. Multi-knapsack problem. The OPL model in figure 1 is taken from [2, p. 26], our second specification below is a direct translation of this model.

```
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];

var int take[Items] in 0..maxValue;
maximize
    sum(i in Items) value[i] * take[i]
subject to
    forall(r in Resources)
        sum(i in Items) use[r,i] * take[i] <= capacity[r];
```

FIGURE 1. Multi-knapsack problem in OPL

Date: January 2004.

¹at our research centre, there is a variation of this problem on every Tuesday. Given n researchers and PhD students, each with a given capacity for sweets and each valuing certain brands, buy as many sweets with a £5 budget such as to satisfy most.

The total number of items are given as `nbItems` and `nbResources`, respectively. These determine the Integer ranges `Items` and `Resources`. The array `capacity` represents the capacities of the resources, the array `value` the value of each item, and `use[r,i]` the number of uses of resource `r` by item `i`. The Integer `maxValue` presents an upper bound on the capacity values and finally the array `take` represents the decision variables. Table 1 shows the optimal solution with objective value 261922 (taken from [2, p. 27].)

<i>i</i>	1	2	3	4	5	6	7	8	9	10	11	12
<code>take[i]</code>	0	0	0	154	0	0	0	913	333	0	6499	1180

TABLE 1. Optimal solution to the multi-knapsack problem of figure 1 on the preceding page

2. Z MODEL

2.1. **Simple knapsack.** We introduce the objects as given set.

[*Objects*]

In the following schema, *volume* and *value* are both part of the instance data, the decision variable is

$ \begin{array}{l} \textit{KP} \\ v : \mathbb{N} \\ \textit{volume}, \textit{value} : \textit{Objects} \rightarrow \mathbb{N} \\ \textit{taken_objects} : \mathbb{F} \textit{Objects} \\ \hline \textit{taken_objects} \subseteq \textit{Objects} \\ \sum(\textit{volume} \parallel \textit{taken_objects}) \leq v \end{array} $

taken_objects. The formulation of the constraints is obvious and uses the Σ operator in combination with ‘bag image’, both defined in the Appendix. We conclude the simple variant with the schema for the optimisation part, which is according to our standard template for optimisation problems.

$ \begin{array}{l} \textit{KP_Optimisation_Part} \\ \textit{KP} \\ \textit{objective} : \textit{KP} \rightarrow \mathbb{N} \\ \textit{solution} : \textit{KP} \\ \hline \forall k : \textit{KP} \bullet \textit{objective}(k) = \sum(\textit{value} \parallel k.\textit{taken_objects}) \\ \textit{objective}(\textit{solution}) = \max(\textit{objective}(\textit{KP})) \end{array} $

Notice that we have included the schema *KP* as well as using it as type, since we need to access the *value* data function.

2.2. **Multi-knapsack problem.** We begin with the instance data, where we first declare those constants and sets that are referenced by other parts of the data later on.

$ \begin{array}{l} \textit{nbItems}, \textit{nbResources}, \textit{maxValue} : \mathbb{N} \\ \textit{Items}, \textit{Resources} : \mathbb{F} \mathbb{N} \end{array} $

We have, for a clearer representation, accommodated all parts of the instance data into the next schema, which also shows how the ranges *Items*, *Resources* are constructed and how *maxValue* is derived.

MKP_Data

$$\begin{aligned} \text{capacity} &: \text{Resources} \rightarrow \mathbb{N} \\ \text{value} &: \text{Items} \rightarrow \mathbb{N} \\ \text{use} &: (\text{Resources} \times \text{Items}) \rightarrow \mathbb{N} \end{aligned}$$

$$\begin{aligned} \text{Items} &= 1 \dots \text{nbItems} \\ \text{Resources} &= 1 \dots \text{nbResources} \\ \text{maxValue} &= \max(\text{ran capacity}) \end{aligned}$$

This leads to a rather short schema for the main problem constraints, where the decision variable is *take*, as

MKP

$$\begin{aligned} \text{MKP_Data} \\ \text{take} &: \text{Items} \rightarrow 0 \dots \text{maxValue} \end{aligned}$$

$$\forall r : \text{Resources} \bullet \text{sumseq}\{i : \text{dom take} \bullet i \mapsto \text{take}(i) * \text{use}(r, i)\} \leq \text{capacity}(r)$$

in figure 1 on page 1. We use the *sumseq* function (cf. Appendix), which sums up the range elements of a sequence. The sequence is explicitly constructed here, using set builder notation within the curly brackets; each *i* from the domain of *take* is mapped into the product $\text{take}(i) * \text{use}(r, i)$. The schema for the optimisation part now concludes the presentation of the multi-knapsack problem.

MKP_Optimisation_Part

$$\begin{aligned} \text{MKP_Data} \\ \text{objective} &: \text{MKP} \rightarrow \mathbb{N} \\ \text{solution} &: \text{MKP} \end{aligned}$$

$$\begin{aligned} \forall m : \text{MKP} \bullet \text{objective}(m) &= \text{sumseq}(\text{value} \otimes m.\text{take}) \\ \text{objective}(\text{solution}) &= \max(\text{objective}(\text{MKP})) \end{aligned}$$

Again, we are using the *sumseq* function to sum up range values in the *objective* function. The inner term of the *sumseq* expression is also a form of sequence arithmetic, multiplying two Integer sequences element-wise (cf. Appendix). This is clearly possible, since the domains of *value* and *take* are the same Integer ranges (namely *Items*). As in the presentation of the simple knapsack problem, we perform schema inclusion in order to reference the function *value* of the instance data.

REFERENCES

- [1] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
- [2] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, January 1999.