The multiple-choise multidimensional knapsack problem

Greedy implementation

Mattia Dell'Oca, Luca Di Bello, Manuele Nolli

1 Problem description

1.1 Mathematical representation

1.1.1 Sets/Domains

- N: Sets of items divided in
- $J = (J_1, J_2, ..., J_n)$: n disjoint classes
- $r_i = |J_i|$: Number of items in each class
- $C = (C^1, C^2, ..., C^m)$: Resource vector of size m (constrained multidimensional capacity of the knapsack)
- $v_{i,j}$: Value of item $j \in \{1..r_i\}$ for class $i \in \{1..n\}$
- $w^k_{i,j}$: Weight of item $j \in \{1..r_i\}$ for class $i \in \{1..n\}$, for the resource $k \in \{1..m\}$

1.1.2 Mathematical model

$$z = \min \sum_{i=1}^{n} \sum_{j=1}^{r_i} v_{i,j} * x_{i,j}$$

$$s.t. \sum_{i=1}^{n} \sum_{j=1}^{r_i} w^k_{i,j} * x_{i,j} \leq C^k \qquad \forall k \in \{1..m\}$$

$$\sum_{j=1}^{r_i} x_{i,j} = 1 \qquad \forall i \in \{1..n\}$$

$$x_{i,j} \in \{0, 1\}$$

1.2 Input data

Data are provided in .txt files, which are divided in *standard* and *large*. Each file represent an independent instance of the MMKP problem and the number of classes, items and weights can change depending on the file. There are a total of 268 standard and 27 large datasets.

$N \quad M$
$Q1 \quad Q2 \quad Q3 \quad ... \quad QM$
I
$V1 \quad W11 \quad W12 \quad W13 \quad ... \quad W1M$
$V2 \quad W12 \quad W22 \quad W23 \quad ... \quad W2M$
...
$VI \quad WI1 \quad WI2 \quad WI3 \quad ... \quad WIM$

- N = Number of classes
- M = Number of resources (weights)
- Q1 .. QM = Capacity of i-th resource

N classes definition follow:
    I = Number of items on the j-th class
    I items definition follow:
        V1 = Value of the item
        W11 .. W1M = Weight of the item for i-th resource

1.3 Output data

Data are outputted to a .out file having the same name of the input file. Foreach execution, a .out file is generated, containing the solution for the specific instance. The content of the file is a one-line vector of item indexes, separated by a whitespace. The total of indexes is N, where N is the number of classes read from the input file.

2 Problem analysis

The given instances have a large number of classes and items. In fact, the CPLEX solver will fail to calculate the optimal value. During the analysis, it was decided to reduce some instances by about half in order to obtain an optimum value using CPLEX solver. The aim was to analyse the result and produce a solution as close to the optimum as possible.

2.1 Instance Converter

Unfortunately, the input data is not compatible with AMPL, so a converter had to be written. The converter *DataConverter.cpp* reads the data in exactly the same way as the algorithm, but writes data in AMPL-readable format to files with the extension .dat in the same folder as the input. Below is an example of output:

```
%%writefile mmkp_a_07_reduced2.dat
param nc := 10;
param nn := 5;
param nr := 5;

# Max capacity
param q :=
  1 72
  2 57
  3 60
  4 65
  5 67
;

# Volume (columns = N = items,  rows = C = classes)
param v (tr): 1 2 3 4 5 6 7 8 9 10 :=
1 129 113 136 129 141 191 228 181 96 203
2 219 219 148 202 173 229 153 208 131 211
3 243 229 194 228 213 234 204 237 220 216
4 262 266 227 265 217 238 216 257 161 235
5 278 274 251 210 190 245 219 257 166 190
;

# Weights
param w := # Resource1 (columns = N = items,  rows = C = classes)
[*,*,1](tr): 1 2 3 4 5 6 7 8 9 10 :=
1 3 0 4 2 1 5 4 7 3 1
2 4 5 0 3 1 9 3 6 1 1
3 9 6 5 8 6 1 7 2 7 5
4 7 3 3 6 6 4 7 4 2 3
5 7 5 9 4 8 7 4 6 0 3
# Resource2 (columns = N = items,  rows = C = classes)
[*,*,2](tr): 1 2 3 4 5 6 7 8 9 10 :=
1 4 2 1 6 3 4 3 2 2 0
2 0 2 4 3 5 3 0 2 1 9
```

```
3 6 5 2 1 4 6 3 2 7 4
4 4 7 5 5 5 2 2 8 7 7
5 5 6 8 8 0 3 6 8 3 9
# Resource3 (columns = N = items,  rows = C = classes)
...
;
```

2.2 AMPL solver

The AMPL model code can be seen below:

```
%%writefile model.mod
param nc;
param nn;
param nr;

# Sets
set C := 1..nc; # Classes
set N := 1..nn; # Items
set R := 1..nr; # Resources

# Parameters
param q{R};     # Resource capacity
param v{C,N};  # Value of each item in each class
param w{C,N,R};  # Weight of each item in each class and resource

# Variables
var x{C,N} binary;   # Whether each item in each class is selected

# Objective
maximize z: sum{c in C, n in N} v[c,n]*x[c,n];


# Constraints
s.t. capacity_constraint{r in R}:
  sum{c in C, n in N} w[c,n,r]*x[c,n] <= q[r];

s.t. class_constraint{c in C}:
  sum{n in N} x[c,n] = 1;
```

2.3 Scripts

In order to speed up and analyse the proposed solutions, 3 scripts were created:

- *Converter*: Allows to easily convert an input file into an AMPL-readable format.
  - *Example*: `./convert-data.sh data/standard/mmkp_a_07.txt`
- *Verifier*: Allows to check if a computed solution is feasible and, if possible, computes the difference between the result obtained and the optimum result.
  - *Example*: `./start-verifier.sh  data/standard/mmkp_a_07.txt`
- *MMKP*: Allows to easily start MMKP on an input file without the need to specify command line parameters manually.
  - *Example*: `./start-mmkp.sh  data/standard/mmkp_a_07.txt`
- *Runner*: All instances are executed and subsequently verified.
  - *Example*: `./runner.sh [--skip-compute] [--only-standard]`
- *Time average*: Calculate the time average of the solutions found.
  - *Example*: `py time-average.py`

3 Solution

The solution is an alternative to the well-known *MMKP relaxation* solution. It works as follows:

- *Class sorting*: based on the average weight divided by the remaining knapsack capacity for each resources.
  - *Ratio calculation*: $ratio = \sum_{c=1}^{m} \frac{\sum_{n=1}^{N} w_{i,n}^{c}}{C^{c}} \forall i \in n$
- *Item sorting*: based on the value, weight and remaining capacity.
  - *Ratio calculation*: $ratio = \sum_{j=1}^{r_i} \sum_{k=1}^{m} \frac{w_{i,j}^{k} -*}{C^{k}} \forall i \in n$

It is important to note that the ratios are based on the remaining capacity of the knapsack. In addition, the class sort is performed to start with the worst class, while the item sort is performed to start with the best item. Finally, the sorts are performed each time an item is selected for a particular class.

4 Pseudocode

Algorithm 1 MMKP greedy algotithm

```
1:  ▷ Main procedure ◁
2:  procedure MAIN(t, input)                                   ▷ t is max time, input is path
3:      ▷ Read file ◁
4:      signal(SIGINT, signalHandler)                          ▷ Register interrupt signal and assign handler
5:      validateParameters(t, input)                           ▷ If t and input are not valid, exit
6:      instance ← readInput(input)                            ▷ O(nClasses * nItems * nResources)

7:      ▷ Compute Greedy solution ◁
8:      sortedClasses ← sortClassByRatioStd(instance.classes)  ▷ quick sort, O(nClasses*log(nClasses))
9:      for i < instance.nClasses do
10:         sortedItems ← sortItemsByRatioStd(instance.classes[i].items)  ▷ quick sort, O(nItems*log(nItems))
11:
12:     for i < sortedClasses do                               ▷ O(nClasses)
13:         classIndex ← sortedClasses[i]
14:         itemTook ← false
15:         for j < sortedItems[i] do                          ▷ O(nItems)
16:             doesItemFit ← true
17:             for k < instance.nResources do                 ▷ Check if item fit in the backpack, O(nResources)
18:                 if sortedItems[i][j].weight[k] > instance.capacities[k] then
19:                     doesItemFit ← false
20:                     break
21:             if doesItemFit then
22:                 instance.solution[classIndex] ← itemIndex  ▷ Add item to the backpack
23:                 itemTook ← true
24:                 for k < instance.nResources do             ▷ Reduce resources, O(nResources)
25:                     instance.capacities[k] ← instance.capacities[k] − item.weights[k]
26:                 break
27:         if !itemTook then
28:             print("No item for class" + classIndex + "fit!")
29:             return 1                                       ▷ Error code
30:     erase(sortedClasses[i])                                ▷ Remove i-th class

31:     writeSolution()
32:
33: ▷ writeSolution procedure ◁
34: procedure WRITESOLUTION                                    ▷ write soluton to output file
35:     total ← 0
36:     for i < instance.nClasses do                           ▷ O(nClasses)
37:         itemValue ← instance.values[i][instance.solution[i]]
38:         total ← total + itemValue
39:     print("Total value : " + total)
40:     open(outfile)                                          ▷ open stream of outfile
41:     for i < instance.nClasses do                           ▷ O(nClasses)
42:         outfile ← instance.solution[i]                     ▷ write to outfile
43:     close(outfile)                                         ▷ close stream
44:
45: ▷ signalHandler procedure ◁
46: procedure SIGNALHANDLER(signum)                            ▷ signum is the number of the received signal
47:     print("Program interrupted with signal" + signum + ", running finalizing code")
48:     writeSolution()
49:     exit signum
```

5 Complexity analysis

The complexity of the algorithm is determined by three parameters:

- **nClasses**: the number of classes in the instance
- **nItems**: the number of items. This value can vary for each class, therefore the worst-case scenario will be considered. In this scenario, nItems is equal to the dimension of the class with more items
- **nResources**: the number of resources in the problem

The operations that have to be considered are those commented in **Pseudocode**. The overall complexity time is determined by:

$$O(readInput) + O(sortClassByRatioStd) + O(sortItemsByRatioStd) + O(generateSolution) + O(writeSolution)$$

Which values are:

$$O(nClasses \cdot nItems \cdot nResources) + O(nClasses \log nClasses) + O(nItems \log nItems) + O(nClasses \cdot nItems \cdot 3 \cdot nResources) + 3 \cdot O(nClasses)$$

The equation can be aproximated as follow:

$$O(nClasses \cdot nItems \cdot nResources)$$

6 Performance analysis

The performance analysis of the MMKP greedy solver revealed that the solver's solution quality was around 80% of the optimal solution for most problem instances. While the solver may be useful for quick approximations (the average time for calculate a solution is 1.2s or initial solutions, its limitations suggest that further optimization is needed.

7 Known limitations

The goal of this algorithm is to run as fast as possible (at best, every instance should have its result computed in less than a second). Greedy algorithms are known to be very fast but to generate solutions which are not optimal, and this one is no expection. In addition, another goal in the development of this algorithm was to always generate a valide solution, always choosing an item for each class without filling the backpack completely. Therefore, the final solution is approximately 80% of the optimal solution, has seen in **Chapter 5: Performance analysis**.

8 Previous attempts

During the development of this project, many solution had been tried. The two more relevant are the following:

- **Max value**: the first (quite naive) attempt was to always take the item with the higher value in each class. The goal of this approach was not to generate an accettable solution (it was obvious that the solution would have violated the constraints almost certaintly) but to familiarize with the problem and its instances. This was crucial for setting up the code structure that would have been used during the rest of the project, changing the criteria of choice.
- **Value/weights mean ratio**: The second attempt had a more consistent goal: finding an accettable choice criteria. After some thoughts, it was decided to use the following mathematical formula:

  $$z = \frac{v_{i,j}}{\sum_{k=1}^{m} w_{i,j}^{k}}$$

  This value was calculated for each item and an average was saved for each class. The class with the max average value/weight mean ratio was considered the best, and therefore it was the first one to chose from.
  The choice for each class was not performed in order, but starting from the classes considered to be the best. Given a class, the item chosen was the one with the highest ratio.

9 Conclusions

Given **Complexity analysis**, it may be appropriate to proceed with a more advanced algorithm, such as a LocalSearch algorithm, to further improve solution quality. LocalSearch algorithms are known for their ability to improve upon initial solutions and explore the search space more effectively, potentially resulting in higher quality solutions. Therefore, by using a LocalSearch algorithm in combination with the MMKP greedy solver, it may be possible to achieve better results for the problem at hand.