The multiple-choise multidimensional knapsack problem

LocalSearch 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 following analysis is based on the greedy algorithm descripted in the previous paper. [1]
Local search algorithms for MMKP face the challenge of a large search space and potential local optima.

3 Solution

The structure of a local search algorithm is always similar, but the size of the neighborhood and the operations to be performed on it change.
The decision made in this project is to compute a single neighbour out of the neighborhood. If the neighbour implements the current solution, than the latter is updated.
Below are three implementations of local Search developed during this project:

- Swap of two items for two classes with others random items [2]
- Swap of one item for one class with another random item [3]
- Swap of one item for one class with the item with highest Value V [4]

The solution proposed is the random double item swap for two different classes, which yielded the best results.
In all the solutions, the stop condition is always false, therefore the program can not stop unless a SIGINT is received.

4 Pseudocode

Algorithm 1 Two Swap LocalSearch

```
▷ Compute LocalSearch solution ◁
procedure COMPUTE
    s ← initialSolution()                                   ▷ Greedy solution
    while True do                                           ▷ O(n)
        N ← computeNeighbor(instance)
        neighborCapacities ← computeCapacitiesFromNeighbor(N, instance)
        ▷ Update current instance with computed neighbor ◁
        instance.solution ← N
        instance.capacities ← neighborCapacities
▷ compute neighbor from instance ◁
procedure COMPUTENEIGHBOR(instance)                         ▷ O(nClasses + nResources)
    neighborhood ← actualSolution
    firstTargetClass ← random
    secondTargetClass ← random
    if firstTargetClass == secondTargetClass then
        change secondTargetClass
    itemFirstClass ← random
    itemSecondClass ← random
    neighborhoodCapacities ← nrResources
    for all r ∈ nrResources do
        adapt swap values for resource r
    feasible ← true
    for all i ∈ neighborhoodCapacities.size do
        if neighborhoodCapacities[i] < 0 then
            feasible ← false
    if feasible & & swapImproveValue then
        execute swap
    return neighborhood
▷ Compute a capacity array from a neighbor solution ◁
procedure COMPUTECAPACITIESFROMNEIGHBOR(neighbor, instance)  ▷ O(nClasses * nResources)
    neighborCapacities ← instance.capacities
    for i ← 1 to instance.nClasses do
        for j ← 1 to instance.nResources do
            if N[i] ≠ instance.solutions[i] then
                neighborCapacities[i] -= weight(N[i])
                neighborCapacities[i] += weight(instance.solutions[i])
    returnneighborCapacities
```

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:

$$O(readInput) + O(greedysolution) + O(compute) * O(computeNeighbor) \\ + O(compute) * O(computeCapacitiesFromNeighbor) + O(Writesolution)$$

Which values are:

$$O(nClasses \cdot nItems \cdot nResources) + O(nClasses \cdot nItems \cdot nResources) + O(n) * O(nClasses + nResources) + O(n) * O(nClasses * nResources) + O(nClasses)$$

The equation can be aproximated as follows:

$$O(n * nClasses * nItems * nResources)$$

6 Performance analysis

The performance analysis of the MMKP local search solver revealed that the solver's solution quality was around 97% of the optimal solution for most problem instances.

6.1 Benchmark

Due to the high time limit of 60 seconds, the benchmark was limited to 10 instance files. The three columns represent:

- **Non Zero Sol.** The total number of solvable instances. The value is calculated on all the available instances and is not limited to the 10 instances used for the banchmark.
- **Mean Delta (%)** The average delta value from the given best value for every instance.
- **Mean Time (s)** The average time required for computing an instance.

	Non Zero Sol.	Mean Delta (%)	Mean Time (s)
`Swap two random items`	258	-2.113	60
`Swap one random item`	258	-3.629	60
`Swap one item with the highest V`	258	-5.834	60

Table 1: Benchmark solutions

7 Previous attempts

The final solution is very close to the one proposed in the paper. [2]
As can be seen in the **Benchmark chapter**, two other solutions, explained in the**Solution chapter** were tested, but performed worse.

8 Conclusions

Although the results are very good, local search algorithms often get stuck in local optima. Therefore, in order to find the global optimum, it would be preferable to implement meta-heuristic algorithms, which are known for their ability to get out of local optimum.

Bibliography

[1] M. Dell'Oca, L. D. Bello, and M. Nolli, *The multiple-choise multidimensional knapsack problem - Greedy implementation*. Apr. 2023.

[2] A. Sbihi, M. Mustapha, and M. Hifi, *Reactive Local Search-Based Algorithm for the Multiple-Choice Multi-Dimensional Knapsack Problem*. Jan. 2007.

[3] S. Iqbal, M. F. Bari, and M. S. Rahman, *Solving the Multi-dimensional Multi-choice Knapsack Problem with the Help of Ants*. 2010.

[4] M. Hifi, M. Michrafy, and A. Sbihi, *Heuristic algorithms for the multiple-choice multidimensional knapsack problem*. 2007.