The multiple-choice multidimensional knapsack problem

Metaheuristic 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

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

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

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

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

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 algorithms described in previous papers. In the first paper [1], the developed greedy algorithm was described and analyzed, which allowed us to quickly find a good feasible solution for MMKP. In the second paper, the LocalSearch algorithm [2] allowed us to find even better solutions using an iterated approach with the concept of Neigbours/Neighbourhood to explore possible more valuable item combinations.
The main disadvantage of LocalSearch is that, by accepting every improving solutions, the algorithm has an high probability to get stuck in a local optima, without exploring other better solutions that could possibily represent the global optima.
To overcome this limitation, we implemented a *simulated annealing* algorithm, which, with a probabilistic approach, accepts worse solutions in order to escape the local optima and explore a wider space of possible solutions. This algorithm has two configurable parameters:

1. The first parameter, called temperature (C), adjusts the probability of accepting worst-case solutions. Initially, the temperature is set to a high value and is gradually reduced over the course of the algorithm.
2. The second parameter, called reduction rate (L), determines the rate of temperature reduction. The choice of this parameter is crucial to the success of the algorithm and must be determined experimentally.

The tuning of these parameters is vital to find high-valued solutions.

3 Solution

Since there is a time limit, it was necessary to implement a memory system for the best solution currently found. Given that in the event of an interruption during the execution of the algorithm, it is necessary to return the best and not the last solution found. To solve the above problem, a copy constructor was created to update the best current solution.
However, the algorithm operates in the current instance, so that it can also select worst-case solutions, in order to get out of a local minimum.
In order to improve the performance of the algorithm, various parameters were tested to find a satisfactory *decrease rate*. In case the time limit is 60 seconds, the intention is that the algorithm can make a wider search for worst-case solutions.
The best value for the variable C turned out to be *600* by means of analyses on several instances of the standard type.

4 Pseudocode

Algorithm 1 MMKP Metaheuristic algorithm

```
▷ Compute LocalSearch solution ◁
procedure Compute
    instance ← initialSolution()                                   ▷ Greedy solution instance
    opt ← instance                                                 ▷ Best instance found
    optValue ← computeSolutionValue(opt)                           ▷ O(nClasses)
    ▷ Simulated Annealing parameters ◁
    C ← 600
    L ← 0.9999
    while True do                                                  ▷ O(n)
        N ← computeNeighbor(instance)                              ▷ O(nClasses + nResources)
        NValue ← computeSolutionValue(N)                           ▷ O(nClasses)
        neighborCapacities ← computeCapacitiesFromNeighbor(N, instance)   ▷ O(nClasses * nResources)
        ▷ Update current instance with computed neighbor ◁
        instance.solution ← N
        instance.capacities ← neighborCapacities
        ▷ Is the neighbor a better solution? ◁
        delta ← NValue − optValue
        if delta >= 0 then
            instance.solution ← N
            ▷ Check if new optimal ◁
            if NValue >= optValue then
                optValue ← optValue
                opt ← N
        else
            probability ← exp(delta/C)
            ▷ Random value between 0 and 1 ◁
            random ← rand(0, 1)
            if random > probability then
                instance.solution ← N
                instance.capacities ← neighborCapacities
            C ← C * L
```

```
▷ 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
▷ Computes the total value of a solution ◁
procedure computeSolutionValue(solution)                     ▷ O(nClasses)
    totalValue ← 0
    for i ← 1 to instance.nClasses do
        totalValue += value(solution[i])
    returntotalValue
```

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:

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

Which values are:

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

The equation can be aproximated as follows:

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

6 Performance analysis

The performance analysis of the MMKP *Simulated Annealing Metaheuristic* solver revealed that the solver's solution quality was around 98.5% of the optimal solution for most problem instances.

7 Known limitations

Using the implemented algorithm, the solutions obtained are more than satisfactory, several instances obtaining values less than 0.5% away from the optimum value. However, problems were encountered with instances of type "b" where the results are around 2%. This could be due to the *descreare rate* and *temperature* parameters setted to obtain the best performance on all types of instances.

8 Previous attempts

We made an attempt to implement the Ant Colony Optimization (ACO) algorithm seen during the Advanced Algorithm course. However, we faced significant challenges in managing the pheromone trails with a Min-Max System (where each pheromone trail has a lower and upper bound) using matrices. As a result, we decided to switch to another metaheuric algorithm.
Given the excelent results with the 2-OPT LocalSearch algorithm described in paper [2], we decided to improve it with the LocalSearch Metaheuristic variations learned during the course. After careful consideration, we chose to implement Simulated Annealing. This decision was motivated by the need to analyze as many solutions as possible and provide a more competitive solution within a reasonable timeframe.

9 Conclusions

We can be satisfied with the results, both in terms of running time and proximity to the optimum. The team has in fact developed three valid algorithms to solve the *Multiple-choise Multidimensional Knapsack Problem* in an approximate manner.

Bibliography

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

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