Lab 2 – In search of the best knapsacks

Group: 17, Paradis, Karl, Larsson, Therése

INTRODUCTION

Multiple knapsack problem

The knapsack problem describes following dilemma: We have a set of items that each have a weight and value. We also have a single container, the knapsack, which we want to fill with items in such a way that the total value of the selected items are as high as possible. However, the knapsack has a limited weight capacity, which the selected items total weight must not exceed.

The multiple knapsack problem is an extension to the problem above as it considers the usage of multiple knapsacks.

Some real life examples of the knapsack problem are efficient use of storage facilities, efficient cargo transports, packing traveling suitcases etc.

Greedy algorithm

One way so solve the knapsack problem would be to use a greedy algorithm which iteratively adds upon a partial solution until a final solution is found. It considers which items to add based on the relative benefit per weight (value divided by weight). When comparing two items, the algorithm chooses the item with the highest benefit per weight.

Neighborhood search

One way to improve the solution found by the greedy approach is by using neighborhood search, which iteratively improves a solution by picking the best neighboring solution if it's better than the current best solution.

DESIGN AND IMPLEMENTATION

Descriptions of the designed and implemented algorithms.

The Greedy Algorithm

The greedy algorithm simply chooses the most beneficial(value/weight) item from a linked list sorted descendingly by benefit. It then adds the item to a knapsack where it fits. This process is repeated until there are no more items in the list or all the knapsacks are full.

Pseudo code for the Greedy Algorithm

```
1. Order the items by their benefit (value/weight) in descending order.
2. Take the top item(the most beneficial) from the item list.
3. Add the item to one of the knapsacks where it fits. If the item doesn't fit in any knapsack,
   then add it to the leftover list.
4. Go to step 2 until the item list is empty or all knapsacks are full.
```

Neighborhood Search Algorithm

When we have a feasible solution found by the greedy algorithm we use that solution to generate neighbors. For every new neighbor that is generated we make a copy of the best feasible so far and then make modifications to it according to the neighborhood definition described below. We then compare the neighboring solution to the best solution so far and choose the most optimal and repeat until we can't find any neighbors with a better solution. This is most likely a local optimal solution.

We used 3 different strategies to define the neighborhood of a solution:

1. The first strategy simply tries to move leftover items into one of the knapsacks where it fits.
2. The second strategy tries to move an item from one knapsack to another. For each knapsack we check if there is any free weight. If there is (freeWeight > 0) then we search in the other knapsacks for the heaviest item thats is less than or equal to freeWeight (item.weight <= freeWeight). If a fitting item is found then it is moved to the knapsack with free weight.

3. The third strategy for each knapsack removes an item to make room for leftover items. We tried 6 different heuristics to select which item to remove. The item with best benefit/worst benefit. The item with highest value/lowest value. The item with highest weight/lowest weight. We could also have tried to select the heaviest item thats is less than or equal to the weight of the most beneficial item in the leftover list, but we did not have time to try this.

Pseudo code for the Neighborhood Search Algorithm

```
1. Generate best feasible solution so far (x) using the greedy algorithm.
2. For each of the neighborhood generation strategies s(), s'(), s''():
        3. Generate a new feasible neighboring solution to x using current strategy s(x).
        4. If value v() of neighboring solution is greater than the best feasible solution so far()
        v(s(x)) > v(x) then set x = s(x) and repeat step 2 starting over with first strategy s().
5. None of the neighboring solutions improves the best feasible solution so far. This means
that x is most likely a local optimal solution.
```

How did it work out?

The greedy algorithm performed much better than expected. The solutions found by it were really quite good. The neighborhood search algorithm on the other hand did not perform so well since it struggles to find neighbors that are improving on the solution found by the greedy algorithm. This could be because of bad neighborhood definition and if we did it again we would try other strategies to generate the neighboring solutions.

CONCLUSION AND FUTURE WORK

Testing the algorithms

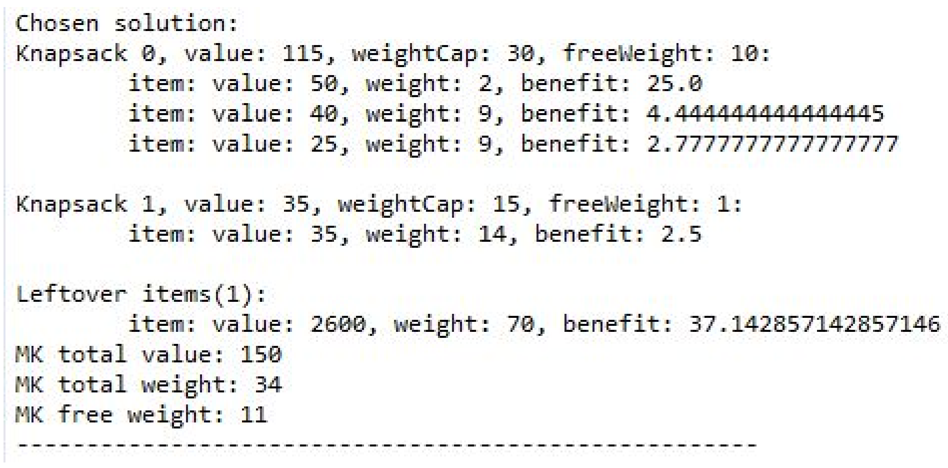
To test the correctness of our algorithms we constructed test-scenarios in which we gave specific value and weight to a set of items and also decide a maximum capacity for e.g. 2 knapsacks. We then evaluated the optimal solution ourselves and then executed the program to see if the implementation reached the same or a similar solution as we did.

Example: Testing the greedy algorithm we considered the following test items:

Value	*Weight*
2600	70
50	2
40	9
25	9
35	14

For this scenario we used 2 knapsacks, with maximum capacity of 30 and 15 respectively. It's easy to see that the first item, with weight 70, won't fit in either of the knapsacks. The first knapsack will then fill up with the following three items, which total weight is now 20 (2 + 2 + 9). The last item won't fit in the first knapsack, since this items weights 14 and the knapsack can now only take a weight of max 10. It will however fit into the second knapsack which has been empty so far with a maximum capacity of 15.

A test run of our greedy algorithm with the same data gives the following solution:



```
Chosen solution:
Knapsack 0, value: 115, weightCap: 30, freeWeight: 10:
        item: value: 50, weight: 2, benefit: 25.0
        item: value: 40, weight: 9, benefit: 4.444444444444445
        item: value: 25, weight: 9, benefit: 2.7777777777777777

Knapsack 1, value: 35, weightCap: 15, freeWeight: 1:
        item: value: 35, weight: 14, benefit: 2.5

Leftover items(1):
        item: value: 2600, weight: 70, benefit: 37.142857142857146
MK total value: 150
MK total weight: 34
MK free weight: 11
-----------------------------------------------------
```

What experiences/knowledge have you acquired?

We have acquired knowledge about the (multiple) knapsack problem and two ways, a greedy approach and a improved neighborhood search approach, to attempt to solve this well known dilemma. We have also become aware of the many different contexts where knapsack problem and other optimization problems could be applicable, both in real life contexts and artificial contexts.

A few sentences on what could have been done differently

We could have tried other definitions of neighboring solutions. And about the implementation we could probably have represented the knapsacks in a more memory efficient way. For example the choice of data structure that stores items is currently a linked list that we keep sorted. This means that for every item that is added to the list the worst-case time complexity is O(N), the good news is getting the best/worst beneficial item is O(1). We could probably have used a priority queue implemented with as a binary heap. This would instead have given us insertion complexity of O(logN) and O(1) for getting the most beneficial item.

What can be improved in your algorithms?

Since the knapsacks problem is NP-hard (Pisinger, 2004), it's reasonable to assume that improvements could be made in several aspects of our algorithms, which applies to the greedy algorithm as well as the neighborhood search. Some of these aspects could be decision making, optimization, switching of items between knapsacks to find better solutions etc. Also more heuristics could be explored to choose which items to switch between knapsacks. Another point of optimization could be to try more complex neighborhood definitions.

Word count: 1198

REFERENCES

- Pisinger, D. (2004). Where are the hard knapsack problems?. *Computers & Operations Research*, *32*(9), 2271-2284. doi:10.1016/j.cor.2004.03.002