Assignment 2-the knapsack problem

Group: 2, Hallberg Sofia, Kareld Oscar

INTRODUCTION

The multiple knapsack problem considers a number of knapsacks with weight restrictions and a number of items with individual weights and values and the task is to fill the knapsacks and maximize the total value of the knapsacks without exceeding the weight restrictions.

An initial solution to the multiple knapsack problem is implemented using a greedy algorithm that builds a solution piece by piece, always choosing the next step maximizing the local benefit. The greedy solution is not expected to generate the optimal solution, but a good approximation [1, p. 140].

The greedy solution is then refined using a neighborhood local search where a number of other solutions, in the neighborhood of the greedy solution, is examined. This means that based on a solution, a small change is made, and if the change generates a better total value the change is kept, otherwise rejected [1, p. 297].

DESIGN AND IMPLEMENTATION

generateRandomData()

To create a large variety of problem instances with various sizes a number (1000) of instances including the number of knapsacks, weight restrictions for the knapsacks, number of items, item values and item weights are randomly generated at execution time. For every instance knapsack-objects are created and put into a knapsack-list and item-objects are created and added to an item-list.

readInput()

To enable verifiability of the solution the readInput-method can be used instead of the generateRandomData-method. The readInput-method reads a few known problem instances from a text file, so the result from the optimization can be verified.

greedyAlgorithm()

The greedy algorithm accesses a list with empty knapsacks and a list with items sorted by descending benefit, where the benefit is the ratio between value and weight.

The algorithm starts with the first knapsack in the knapsack-list and fills it with items from the item-list, in the order of descending benefit, until the weight limit of the knapsack is reached. Then the next knapsack in the knapsack-list is filled with items from the item-list, starting with the first available item in the list continuing with available items with descending benefit. The procedure is continued until all items are placed in a knapsack or it is not possible to include any of the available items in any of the knapsacks because of the weight limitations.

Pseudo code

```
empty all knapsacks;
set all items to available;
for (all knapsacks)
    for (all items in the list, sorted from highest benefit to lowest)
        if (item is available AND item weight is less than free space in the knapsack)
            add item to knapsack;
            set item availability to false;
```

neighbourhoodSearch()

The neighbourhoodSearch accesses the greedy solution as a list with knapsack-objects where every knapsack contains a list with item-objects matching the items in the knapsack and the list with items in the order of descending benefit.

The neighborhood is defined as a solution where one item is removed from one knapsack, *knapsackA*, and put into another knapsack, *knapsackB*. (For this item move to be possible there must be enough capacity left in knapsackB.) A new item, not previously included in any knapsack, is then added to knapsackA, if there are any items for which this is possible due to the weight restriction of knapsackA. If the new neighbor solution has a greater value than the current solution the neighbor solution is set as the current solution. The value is the total value for all knapsacks.

The algorithm terminates when the item lists in all knapsacks are iterated once to check if the item can be put in any other knapsack.

Pseudo code

```
save initial total value as an integer;
for (all knapsacks)
    for (all items in the knapsack)
        for (all other knapsacks)
            if (an item from the first knapsack fits inside the second)
                add item to the second knapsack;
                remove item from the first knapsack;
for (all available items)
    for (all knapsacks)
        if (item fits in knapsack)
            add item to knapsack;
check if new total value is higher than initial total value;
```

RESULTS

The greedy optimization performs quite well and to find problem instances with further improvement possibilities after the greedy optimization a large number of instances must be generated and chosen. It also turned out to be significant how the input values to the problem instances are chosen. A problem instance with a relatively large number of items with small weights tends to be more easily optimized in the greedy optimization than a problem instance with fewer items with larger weight.

CONCLUSION AND FUTURE WORK

To show that our neighborhood search algorithm worked as intended, we created a file with some test inputs, with few items and knapsacks and easily calculable benefits. Our greedy search, while effective, sorts the items based on benefit rather than actual value. This is a good starting solution - since another sorting criteria almost always would result in a worse initial result. Our test inputs results in a state where the first knapsack contains an item that would fit inside the other two knapsacks. When found, the item is moved and the algorithm then checks if the freed up space in the first knapsack is enough for any of the items in the list of currently unused items. In the example with our test inputs, the last unused item (with a weight of 3) perfectly fits into the first knapsack.

Regarding what could have been done differently, the definition of a neighbor in our solution could be redefined. For example, we could generate a new neighbor by trying to remove two or more items from a knapsack and replacing them with a new item. If the value of the new item is higher than the total value of the two previous items, the change is beneficial - but the greedy algorithm would not have found it if the respective benefit of the initial items were higher than that of the new item.

The assignment has given us a deeper understanding of how to implement a search algorithm that does not have a known optimal solution. The fact that we ourselves had to define a neighborhood was often more of a hindrance than a blessing. It can be quite frustrating trying to implement an algorithm without having any idea about its effectiveness.

Regarding possible improvements, we believe there are quite a few to be found. First and foremost, the time complexity of our algorithms are far from optimal. We might have been able to see an improvement here if more time were spent on evaluating different data structures.

Another possible improvement is to actually exchange one or more items with each other during the neighborhood search. You could try to replace one item from one knapsack with two or more from another (or from the list of unused items). To implement this, you would need to save an instance of each state - to be able to find the optimal one after you've generated all neighborhoods in the search. This would increase both the time and space complexity but would probably also be able to improve the initial total value more than our current solution.

REFERENCES

[1] Dasgupta, S., Papadimotiou, C. & Vazirani, U. (2006). *Algorithms*. McGraw-Hill Education.