BTH004 - Laboratory assignment 1

In this laboratory assignment you should design and implement different algorithms for the multiple knapsack problem. The assignment contains two parts; one is mandatory and one is optional.

In part 1 (the mandatory part) you should design and implement two algorithms for the multiple knapsack problem:

1. A (contructive) greedy algorithm.
2. An improving search (neighborhood search) algorithm.

Part 2 (the optional part) concerns implementing a tabu-search algorithm (a meta-heuristic) for the considered problem.

The assignment should be conducted individually, and it will be examined through a written report (one per student). The report should fulfill the requirements specified below, and it should be sent to Professor Zhenbo Cheng. The deadline for submitting the report is announced by Zhenbo Cheng.

Note that you are allowed to use any high-level programming language you find suitable, e.g., java, c, c++, or python.

It is highly recommended that you start working with the assignment as soon as possible. It is of particular importance that you start before the scheduled supervision time, so that you can make as much use as possible of the teacher support.

The knapsack problem

The (standard) knapsack problem, which was introduced earlier in the course, can be described in the following way. Assume that you have a set of items, each with a (positive) weight and a (positive) value, and a knapsack (or bag) with a limited weight capacity. The problem is to choose (to include in the knapsack) items so that the weight capacity of the knapsack is not exceeded and the value of the chosen items are as high as possible.

Mathematically, the problem can be formulated in the following way. We let $I = \{1, \ldots, n\}$ be an index set over the n items, where item $i \in I$ have a value $p_i > 0$ and a weight $w_i > 0$, and we let $W > 0$ denote the weight capacity of the knapsack. For item $i \in I$, the binary decision variable x_i is used to determine whether to include item i in the knapsack: $x_i = 1$ if the item is chosen, and $x_i = 0$ if it is not chosen.

The objective function of the problem is to maximize the utility of the chosen items, i.e.,

$$\text{Maximize} \sum_{i \in I} p_i x_i.$$

All valid solutions to the problem should fulfill the weight constraint

$$\sum_{i \in I} w_i x_i \leq W,$$

and the values of the decision variables are restricted by the constraint set

$$x_i \in \{0, 1\} \quad \forall i \in I.$$

The multiple knapsack problem

The multiple knapsack problem is an extension to the standard knapsack problem in that it considers choosing items to include in m knapsacks (the standard knapsack problem considers only one knapsack).

Mathematically, the multiple knapsack problem can be formulated as follows. We let $I = \{1, \ldots, n\}$ be an index set over the n items, where item $i \in I$ have a value $p_i > 0$ and a weight $w_i > 0$. In addition, we let $J = \{1, \ldots, m\}$ be an index set over the m knapsacks, where $W_j > 0$ denotes the weight capacity of knapsack $j \in J$. For item $i \in I$ and knapsack $j \in J$, we let the binary decision variable x_{ij} determine whether to include item i in knapsack j: $x_{ij} = 1$ if item i is included in knapsack j, otherwise $x_{ij} = 0$.

The objective function of the problem is to maximize the utility of the chosen items, i.e.,

$$\text{Maximize} \sum_{i \in I} \sum_{j \in J} p_i x_{ij}.$$

For each of the knapsacks, the solution space is restricted by a weight capacity constraint, which states that the total weight of the selected items for that knapsack is not allowed to exceed the weight capacity of the knapsack. This is modeled by the following constraint set (one constraint for each of the m knapsacks):

$$\sum_{i \in I} w_i x_{ij} \leq W_j, \quad j \in J.$$

In addition, it needs to be explicitly modeled that an item is not allowed to be included in more than one of the knapsacks. This is modeled by the following constraint set (one constraint for each of the n items):

$$\sum_{j \in J} x_{ij} \leq 1, \quad i \in I.$$

Finally, the values of the decision variables are restricted by the constraint set

$$x_{ij} \in \{0, 1\}, \quad i \in I, j \in J.$$

Part 1 - Greedy and neighborhood search algorithms

In the first part (the mandatory part), you should design and implement 1) a greedy algorithm and 2) a neighborhood search algorithm for the multiple knapsack problem.

Greedy algorithm

As discussed earlier in the course, a greedy algorithm is an algorithm that starts with an empty solution, and iteratively adds solution components to partial solution until a complete solution is found.

For knapsack problems, it is possible to construct greedy algorithms that are based on the relative benefit per weight unit for the considered items. We let $b_i = \frac{p_i}{w_i}$ denote the relative benefit per weight unit for item $i \in I$. By comparing $b_{i'} = \frac{p_{i'}}{w_{i'}}$ and $b_{i''} = \frac{p_{i''}}{w_{i''}}$ for two items $i', i'' \in I$, it is possible to make a greedy decision on which item to include. If $b_{i'} > b_{i''}$ (i.e., $b_{i'}$ gives higher value per weight unit than $b_{i''}$) it seems better to choose $b_{i'}$ than $b_{i''}$ for some of the m knapsacks. On the other hand, if $b_{i'} < b_{i''}$ it seems better to include $b_{i''}$ than $b_{i'}$ in some of the m knapsacks. If $b_{i'} = b_{i''}$, it does not matter which one to choose, since $b_{i'}$ and $b_{i''}$ in this case gives the same value per weight unit.

A greedy algorithm for the multiple knapsack problem could be designed by iteratively adding to some knapsack the item with highest relative benefit (i.e., $i' = \arg\max_{i \in I}(b_i)$, such that i' has not already been included in an earlier iteration).

Please note that there are algorithm details not discussed above, which you need to define yourself[1]. For example, termination criteria and choice of knapsack in case there is sufficient amount of capacity in more than one of the knapsacks, has not been discussed above. In addition, you need to consider what you should do if there is no capacity for the item with highest relative value, but there are other items with smaller weight that can be added.

Improving search (neighborhood search) algorithm

Improving search heuristics are algorithms that iteratively improve a feasible solution by doing small modifications of the most recent solution.

For a solution $\overline{x}$, we define a neighborhood $N(\overline{x})$ as all solutions "close enough" to $\overline{x}$ (according to some criteria). If $\overline{y} \in N(\overline{x})$, then $\overline{y}$ is a neighbor of $\overline{x}$.

Neighborhood search algorithms are improving search algorithms where a current solution is iteratively updated by choosing the "best" solution in the neighborhood of the current solution. An algorithm description (for a maximization problem) is provided below.

Step 0: Find a feasible (starting) solution $x^{(0)}$ with value (cost) $c(x^{(0)})$ and set solution index $t = 0$.

Step 1: Determine (all points in) neighborhood $N(x^{(t)})$.

Step 2: If $c(x^{(t)}) \geq c(x) \, \forall \, x \in N(x^{(t)})$, then break with local optimal solution $x^{(t)}$.

Step 3: Choose $x^{(t+1)} \in N(x^{(t)})$ such that $c(x^{(t+1)}) \geq c(x^t)$
Set $t = t + 1$ and goto step 1.

As starting solution to the neighborhood search algorithm, you could, for example, use the solution obtained by your greedy algorithm.

[1] It is typically a good idea to test different approaches to see which performs best

A challenge is how to determine an appropriate neighborhood for the multiple knapsack problem. If the neighborhood is too large, it will be impossible to iterate over all solutions in the neighborhood, if it is too small, there is a high risk that the algorithm will get stuck in a local optima that is not so good.

A possible way to define a neighborhood is to rotate items. As long as you have at least one item that is not included in any knapsack, you could define the neighborhood of a solution as all other feasible solutions that can be obtained by:

- Moving some item (i') from one knapsack (m') to another knapsack (m'').
- Moving some other item (i'') away from knapsack (m'') (in order to make room for item i'). In the obtained solution, i'' is not included in any of the knapsacks.
- Moving some other item (i'''), which is not included in any knapsack in the current solution, into knapsack m'.

If $b_{i'''} > b_{i''}$ a better solution has been found.

When using this type of neighborhood, it is important to consider that it might be possible to, via rotations, make room for an additional item in some of the knapsacks after some item has been replaced by some other item. This needs to be explicitly included in the neighborhood definition.

You are encouraged to identify some other neighborhood that you find appropriate for the considered problem. For example, you might find it interesting to consider rotation involving more than 3 items; however you need to consider that this increases the search space significantly.

Part 2 - Tabu-search

A problem with the neighborhood search heuristics is that they eventually will get stuck in a locally optimal solution. A locally optimal solution is a solution that cannot be improved by moving to any of the solutions that are sufficiently close in the search space (i.e., in the neighborhood), but there are solutions farther away that are better.

The ideas of tabu-search methods, which are based on neighborhood search, are:

- It is possible to move to a solution with worse objective if there is no improving solution in the neighborhood (note that it is necessary to keep track of the so far best found solution).
- The k most recent solutions are tabu, i.e., they cannot be visited (otherwise the algorithm might immediately go back to the locally optimal solution).

Tabu search algorithms (for maximization problems) can be described in a general way using the following algorithm description.

Step 0: Find a feasible (starting) solution $x^{(0)}$ with value (cost) $c(x^{(0)})$ and set solution index $t = 0$.

Step 1: Determine breaking criteria. For instance, break if maximum number of iterations has been reached, or no non-tabu solutions exists.

Step 2: Determine neighborhood $N(x^{(t)})$.

Step 3: Choose $x^{(t+1)} \in N(x^{(t)})$ which is not in the tabu-list, and which maximizes $c(x)$.

Step 4: Update tabu-list (remove the oldest solution in the tabu-list and add the most recent solution, i.e., $x^{(t)}$), set $t = t + 1$ and go to step 1.

In part 2 (the optional part) you are encouraged to extend your neighborhood search algorithm into a tabu-search algorithm.

Requirements on report

The report should contain (at least):

1. Your name.
2. Descriptions of the designed and implemented algorithms, including design choices, such as neighborhood definition, tabu list length, termination criteria, etc.
3. Pseudo code for each of the developed algorithms.
4. Description of how you "showed" the correctness of your algorithms; in particular you need to describe which test cases you used when testing your algorithms.