TD1: Local Search

Instructions:

- You should do every exercise except the hard exercises marked with (⚙).
- Fully read the exercise statement before starting to solve it, and always justify your answers.
- If an exercise asks for an algorithm, describe it in pseudo-code or Python.

Exercise 1: The knapsack problem. In the knapsack problem, we have a bag with limited space, and want to pack items into the bag so as to carry as much value as possible. For each of the n items, we are given its weight w_i and its value v_i, and we are also given the capacity of the bag B (in weight). Our goal is to select a subset of the items such that the total weight of the selected items is at most B, and the total value is maximized.

1. Describe a solution representation for the problem.
2. Assume that the number of items is very large compared to the number of items in a solution. Suggest a more memory-efficient representation for this case, and compare it to the solution of the previous exercise.
3. Would a random search algorithm be useful for this algorithm? Justify.
4. One possible neighborhood operator takes an item that is not in the solution and adds it in. What are the problems with this approach?
5. Suggest a different operator that does not have the same problem.

Exercise 2: Local search for maze solving. In this exercise we will consider the maze solving problem defined in the lecture.

1. Explicitly define a fitness function that first helps the algorithm arrive at a path from the start to the end, and then minimize the length of the path. It should be clear from your description (whether in text or as a formula) what is the fitness value of any solution.
2. Suggest an algorithm to compute the fitness function and analyze its worst-case running time.

From now on, consider the operator that changes the arrow in a cell to a different direction.

3. Describe the neighborhood obtained from this operator explicitly, using only properties of the solution.
4. Does local search always return a feasible solution under this operator?
5. How many iterations might a local search algorithm need to reach a (local) minimum?
6. (⚙) Suggest a local search algorithm (neighborhood and fitness) that does not have any infeasible local minima.

Exercise 3: Bipartite matching. The maximum bipartite matching problem concerns the situation where we have two different sets, and want to create pairs of one element from each set, and to have as many as possible. For example, we can think of assigning medical students to hospital internships, professors to courses, or even football players to positions.

The problem is modeled as a bipartite graph, $G = (A \cup B, E)$, where edges are always between an element of A and an element of B, that is, $E \subseteq \{ab : a \in A, b \in B\}$. A matching $M \subseteq E$ is a subset of the edges that do not share any endpoints, that is, each vertex is incident to at most one edge. Our goal in the maximum bipartite matching is to find a largest matching $M \subseteq E$.

1. Describe a local search algorithm for the maximum bipartite matching problem. You should describe the solution representation, neighborhood, and fitness.
2. A possible operator is to take an edge e in the matching and another edge f that is not in the matching, and to exchange them, that is, remove e from the matching and add f instead. What is the problem with this operator?
3. Another natural operator simply adds an edge to the matching if both of its endpoints are free. We call this the *single-edge* operator. Unfortunately, this does not always lead to a maximum matching. Can you find an example of this?

We can generalize the operator above to get the 2*-edge* operator: given two edges a_1b_1 and a_2b_2 such that a_1 and b_2 are free and a_2b_1 is an edge in the matching, the operator removes a_2b_1 from the matching and adds a_1b_1 and a_2b_2 instead, increasing the size of the matching by 1.

4. Find a graph where the single-edge operator does not achieve the maximum matching, but using both the single-edge and 2-edge operators leads to the optimum solution.
5. How many neighbors (in the worst case) are there under the single-edge and 2-edge operators?
6. Design an algorithm that runs in time $O(n + m)$ and finds a pair of edges for the 2-edge operator (if it exists).

We can further generalize the 2-edge operator to get the *alternation* operator, which takes a sequence of edges $(a_1b_1, a_2b_2, \ldots, a_kb_k)$ of any length such that a_1 and b_k are free, and each edge b_ia_{i+1} is in the matching; it then removes every edge b_ia_{i+1} from the matching and adds the edges a_ib_i instead, which increase the size of the matching by 1.

7. (⚙) How many neighbors (in the worst case) are there, if we use the alternating operator?
8. (⚙) Design an algorithm that runs in time $O(n + m)$ and computes a sequence of edges for the alternating operator, if it exists.

Although we will not see proof of it here, local search with the alternating operator is equivalent to a known algorithm to compute a maximum bipartite matching.