Artificial Intelligence and Knowledge Engineering
Assignment 1

Michał Karol, Piotr Syga

February 28 2023

1 Excercise goal

The purpose of the exercise is to practice optimization problems and implement methods discussed in the lecture for solving a subclass of those problems. After completing the list, the student should know what an optimization problem is, what difficulties may be involved in obtaining an exact solution of the presented problem, and how to deal with solving the problem with limited resources (e.g. computing power or time). In particular, the differences between an approximation and heuristic should be known, as well as examples of heuristic approaches to search problems, and path finding.

2 Theoretical introduction

In this section you will find information useful during solving the stated problems. It is assumed, that after finishing the exercise students are fully knowledgeable about theoretical concepts described below.

2.1 Definitions

Definition 1 (Optimization problem)**.** *Optimization problem is a set of constraints (given in the form of inequalities or equalities with the decision variables) with the objective function. Let S be the feasible set in the assumed problem, i.e., the set of such S that satisfy all constraints, and $f(\bar{x}) : \mathbb{K}^n \to \mathbb{K}$ function to evaluate the quality of the solution. We define the minimization problem finding such a $s^* sin S$ for which $s^* = \min_s f(s)$ and $f(s)$ is a cost (or loss) function. Alternatively, a problem is called maximizing if the goal is to find such a $s^* \in S$ for which every $s^* = \max_{s \in S} f(s)$, where $f(s)$ is a utility function.*

One may easily note that every minimization problem could be transformed into maximizing problem.

Definition 2 (Optimal solution)**.** *Let S be a feasible set for the (minimization) problem $\mathcal{P}$ with cost function f. Solution $s^\star \in S$ is a global optimum* ***iff*** $\forall_{s \in S} f(s^\star) \leq f(s)$.

Definition 3 (Approximation)**.** *Let* $\mathrm{Opt}(x) = \min_{s \in S(x)} f(s)$. *For algorithm M solving optimization problem ($M(x) \in S(x)$), we call it ε–approximation for $\varepsilon \geq 0$, if and only if for each x we have*

$$\frac{|f(M(X)) - \mathrm{Opt}(x)|}{\max\{\mathrm{Opt}(x), f(M(x))\}} \leq \varepsilon \ .$$

Definition 4 (Heuristic)**.** *Let P be an optimization problem, and S a set of solutions for problem P. Let $f : S \to \mathbb{R}$ be a cost function. Let s^* be a globally optimal solution.*

Heuristic H for problem P is a function $H : S \to S$ returning solution $s' \in S$ for problem P, such that for most cases solution s is good enough, meaning that difference $|s^* - s'|$ is not significant.

Heuristics could use simplifications, approximations, and empirical rules in order to reduce computational complexity or in order to increase the chances of finding a good solution in a relatively short time.

A formal definition of heuristic implies that heuristics are not exact algorithms, and are only methods of searching for a good approximate solution to the problem. Heuristics could be used on their own or in conjunction with exact algorithms to improve the performance and quality of found solutions. Heuristics could be used for non-aproximximable problems.

Heuristics are applied in order to solve the traveling salesman problem (TSP), where the target is to find the shortest closed path connecting all cities out of the set of V.

Definition 5 (Traveling salesman problem (TSP))**.** *Let $G = (V, E)$ be a graph, where V is a set of nodes (cites to be visited) with the size of n, and E is a set of edges connecting a pair of cities. Each edge $(u, v) \in E$ has non-negative cost $d(u, v)$ representing distance between cities u i v.*
The goal is to find the shortest closed path that visits each city exactly once and then returns back to the starting point. Formally, one has to find a permutation $\pi = (v_1, v_2, \ldots, v_n)$ of set V, such that sum of edges costs $D = d((\pi_1, \pi_2)) + d((\pi_2, \pi_3)) + \cdots + d((\pi_{n-1}, \pi_n)) + d((\pi_n, \pi_1))$ is minimal.

2.2 A* search algorithm

A* algorithm is a heuristic algorithm applied to search for the shortest path on a graph. The algorithm is an extension of the Dijkstra algorithm, using an additional approximation estimation module from the node to the target. Note that, with heuristics that are not properly designed, the algorithm can return results far from the optimum. Algorithm was proposed in 1968 by Peter E. Hart, Nils John Nilsson and Bertram Raphael.

A* search algorithm is based on the minimization of a cost function defined as $f(curr, next) = g(curr, next) + h(next, end)$, where $g(curr, next)$ is a cost function of going to the node $next$ and $h(next, end)$ estimates cost from $next$ node to the end node.

Commonly used cost estimations:

- Manhattan distance: $h(curr, next) = |curr.x - next.x| + |curr.y - next.y|$
- Euclidean distance: $h(curr, next) = \sqrt{(curr.x - next.x)^2 + (curr.y - next.y)^2}$

The main idea behind the A* algorithm is using of two lists: open - consisting of nodes that were visited, but not all neighbors were visited, and closed list where all nodes on the list were visited as well as all the neighbors.

Algorithm 1 A* algorithm

Input: start, end
Output: List of nodes and edges creating the found path

```
1:  start.g ← 0
2:  start.h ← 0
3:  start.f ← start.g + start.h
4:  open ← list([start])
5:  closed ← list()
6:  while len(open) > 0 do
7:      node ← null
8:      node_cost ← +Inf
9:      for test_node in open do
10:         if f(test_node) < node_cost then
11:             node ← test_node
12:             node_cost ← f(test_node)
13:     if node == end then
14:         Solution found
15:     open ← (open − node)
16:     closed ← (closed + node)
17:     for next_node in neighbourhood(node) do
18:         if next_node not in open and next_node not in closed then
19:             open ← (open + next_node)
20:             next_node.h = h(next_node, end)
21:             next_node.g = node.g + g(node, next_node)
22:             next_node.f = next_node.g + next_node.h
23:         else
24:             if next_node.g > node.g + g(node, next_node) then
25:                 next_node.g = node.g + g(node, next_node)
26:                 next_node.f = next_node.g + next_node.h
27:                 if next_node in closed then
28:                     otarte ← (open + next_node)
29:                     closed ← (closed − next_node)
```

Dijkstra's algorithm Dijkstra's algorithm is a shortest-path search algorithm for a non-negative weighted graph with a single source. The algorithm starts with a set of shortest distances from the source to the nodes and updates it while visiting nodes.
Let $G = (V, E)$ be a weighted graph with a source s, set of nodes V, and set of edges E. Let $w : E \to \mathbb{R}$ be the cost function for traveling the edge. For each $v \in V$, let $d(v)$ be the cost of the shortest path from s to v, and $p(v)$ be a function returning the preceding node of v for the currently shortest path from s to v.

Dijkstra's algorithm works in the following way:

1. Initialize $d(s) = 0$ and $d(v) = \infty$ for each $v \in V \setminus \{s\}$.
2. Create set Q containing all nodes of graph G.
3. While Q is not empty, do:
   (a) Select node $u \in Q$ with smallest $d(u)$ and remove it from Q
   (b) For each v such that $\exists_{(u,\, v)\in E}\ d(v) > d(u) + w(u,\, v)$ update $d(v) = d(u) + w(u,\, v)$ and $p(v) = u$
4. Return d and p

Set Q represents the set of nodes, that are not yet visited, hence with the distance from s not yet updated. In step 3a, the node from Q with smallest value $d(u)$ is selected, and added to the set of nodes with shortest distance from the source. Furthermore, the shortest distances for all neighbors of the node u are updated, if $d(u) + w(u, v)$ is smaller than already stored $d(v)$. This way algorithm finds the shortest path from s to every node in a graph.

2.3 Local search and Tabu Search

Local search The local search algorithm is a metaheuristic that involves an iterative search in the neighborhood of the current solution, with the intention of finding a better solution.

Let P denote the optimization problem and s be the current solution. Let $N(s)$ be the neighborhood of s, that is, the set of solutions that can be achieved by changing a fixed number of parameters (e.g., a different evaluation of one of the variables) of the current solution s. Let $f(s)$ denote the value of the objective function for the solution of s.

The algorithm starts from generating the initial solution s_0. Then, at each iteration, the algorithm generates a new solution $s'(s)$. If $f(s') < f(s)$, the algorithm accepts the new solution s' and sets it as the current solution s. Otherwise, it rejects the new solution and continues searching the neighborhood. The search ends when the stop criterion, which is one of the parameters of the heuristics (e.g., the maximum number of iterations, the maximum number of iterations without updating s), is reached. The local search algorithm is sensitive to getting stuck in the local optimum of the objective function.

Tabu search Tabu search algorithm is a metaheuristic that involves iteratively searching the neighborhood of the current solution, taking into account the set of moves that are not allowed (tabu). Compared to local search, it requires additional memory but eliminates the retesting of the same (or similar) solutions to get out of the local optimum.

The main steps of Tabu search:

1. We generate the initial solution s_0.
2. We set the current solution s as s_0, the tabu set as $T = \emptyset$ and the best known solution as $s^* = s$.
3. We define the neighborhood $N(s)$.
4. Until we meet the stop condition:
   (a) We generate the neighborhood of the current solution $N(s)$.
   (b) We test the whole neighborhood or (deterministically) sample $N(s)$.
   (c) We select the best of the neighbors, such that $s_i \in N(s) \setminus T$.
   (d) $T = T \cup N(s)$
   (e) We set $s = s_i$, if $f(s) \leq f(s^*)$, then we set $s^* = s$.
5. We return the solution s^*.

It can be noted that depending on the presented optimization problem P, the Tabu search algorithm should be adapted to the specific conditions of the task. An example is the time-consuming nature of searching $N(s)$, which, for example, for points in a sphere of radius r is infinite. In such a case, a sampling (e.g., deterministic or Monte Carlo method) of solutions lying in the neighborhood of s is performed. In addition, it is easy to see that the array of forbidden solutions T increases with the number of steps, requiring more and more memory. The basic method of solving this problem is to set the size of the array $t = T\,|T|$ and treat it as a FIFO queue, where in case of overflow the oldest entry is removed. The choice of t should depend on the characteristics of $N(s)$, since t that is too small for large neighborhoods makes it easy to create cycles of length greater than t, causing the same solutions to be revisited, thus problems similar to those of local search. On the other hand, too large t for small neighborhoods results in the consumption of redundant memory and the failure to remove solutions from T. In order to admit solutions from T that, due to sampling, ended up in the array of forbidden solutions without checking the value of the utility function, the aspiration function (or array) A can be used, which specifies that if a solution is good enough, it is considered as a possible solution despite its presence in T.

It is worth noting that we do not always have to keep the list of complete solutions in T. Using the specifics of the problem (or representing the set of solutions as an n-dimensional cube), we can modify T:

- we can prohibit the vertices – instead of prohibiting the entire solution, we prohibit a fixed value at one of the coordinates, e.g., in TSP we prohibit all solutions that have a city j at position i or prohibit changing the position of a city i.
- we prohibit the edges – this approach prohibits movements inside a fixed plane (two or more coordinates), e.g., for TSP in 2-opt prohibits the removal of one of the edges.
- we prohibit the edges with one of the vertices – we forbid moving along a fixed straight line (for two coordinates) or inside a fixed plane (more coordinates), e.g., for TSP, all edges that leave a fixed vertex i are forbidden.

Below, an example of a pseudocode that uses Tabu search for solving traveling salesman problem is shown.

Algorithm 2 Knox algorithm: Tabu search for TSP

```
1: k ← 0
2: randomly select the starting solution s
3: s* ← s
4: T ← ∅
5: while k < STEP_LIMIT do
6:    i ← 0
7:    while i < OP_LIMIT do
8:       define N(s), A
9:       select the best s' ∈ (N(s) \ T) ∪ A
10:      update T
11:      if D(s') < D(s) then
12:         update the local optimum s ← s'
13:      i++
14:   k++
15:   if D(s) < D(s*) then
16:      s* ← s
```

Knox's algorithm allows us to consider a neighboring solution that is in the Tabu list if it is good enough. In doing so, it uses an Aspiration (A) array.

An example of an Aspiration array could be implemented by remembering the history of k last steps of an algorithm. In such a case, in addition to the table T, let's store a history H describing the number of modifications to the variable in h last steps. Let's set $0 < varepsilon < 1$ and $A_i = f(s_i) + varepsilon(k - H(i))$. If $f(s) > A_i$, we select the neighbor s_i.

3 Materials

1. File `connection_graph.csv` contains the transformed data from timetables for Wroclaw's public transport as of 1st March 2023. Data was do-

wnloaded using open-source information provided by the Municipality of Wroclaw. (Open data Wrocław). The file contains the following columns:

(a) id - the identification number of an edge

(b) company - name of the transportation company

(c) line - line name

(d) departure_time - departure time formatted HH:MM:SS

(e) arrival_time - arrival time formatted HH:MM:SS

(f) start_stop - start stop

(g) end_stop - end stop

(h) start_stop_lat - start stop latitude in WGS84

(i) start_stop_lon - start stop longitude in WGS84

(j) end_stop_lat - end stop latitude in WGS84

(k) end_stop_lon - end stop longitude in WGS84

4 Tasks

1. With the use of provided file `connection_graph.csv` implement algorithm searching for shortest paths between given stops `A` and `B`. As a cost function use (depending on the user's decision) travel time from `A` to `B` or the number of transfers needed.

   Application should accept input in form of 4 variables:

   (a) the starting stop `A`

   (b) the ending stop `B`

   (c) optimization criterion: value t denotes travel time minimization, value p means minimization of the number of transfers

   (d) start time

   The solution should print out to the standard output in successive lines detailed path information including starting stop, ending stop, used line's name, starting time, ending time, and on the standard error output the value of minimized criterion and time required for calculation of the shortest path.

   Scoring:

   (a) shortest path search from `A` to `B` using Dijkstra's algorithm, based on travel time criterion (10 points).

   (b) shortest path search from `A` to `B` using A* algorithm, based on travel time criterion (25 points).

(c) shortest path search from `A` to `B` using A* algorithm, based on the number of transfers criterion (25 points).

(d) modifications of A* algorithm from (b) or (c), allowing the reduction of computational time or improving quality of found solution (10 points)

2. Using the provided file `connection_graph.csv` implement the shortest path from a given stop `A` that visits all the stops given in a list `L`= $A_2, \ldots, A_n$, and returns back to `A`. As a cost function use (depending on the user's decision) travel time from `A` to `B` or the number of transfers needed.

   The application should accept input consisting of 4 variables:

   (a) starting stop `A`

   (b) list of stops to visit separated by a semicolon `L`

   (c) optimization criterion: value t denotes travel time minimization, value p means minimization of the number of transfers

   (d) starting time

   The solution should print out to the standard output, in consecutive lines, the detailed path information including starting stop, ending stop, used line's name, starting time, ending time, and on the standard error output the value of the minimized criterion and time required for calculation of the shortest path.

   Scoring:

   (a) an algorithm searching for the shortest path between the nodes based on Tabu Search without bounding the size of T (10 points)

   (b) modification of the algorithm (a) with a variable T size, based on the length of `L`, in order to minimize the cost function (5 points)

   (c) modification of the algorithm (a) by adding Aspiration criterion, in order to minimize the cost function (5 points)

   (d) modification of the algorithm (a) by adding a neighbor sampling strategy to minimize the cost function and reduce the computation time. (10 points)

For each of the tasks prepare a report containing the theoretical background of the used algorithms, an example of using the method, the description of added modifications to the base algorithms, and additional materials used in order to fulfill the task as well as a short description of the libraries used in the implementation. In summary, describe the problems encountered during the implementation. Reports have to be sent to the tutor at least 24h before the tasks' submission.

5 Bibliography

- An interactive animation describing A*
- Forbot's presentation on A*
- Blog post concerning A* and its steps
- A video describing A*
- The original paper on A*
- E. Taillard – a survey on Tabu search
- Knox – Tabu search for a symmetric TSP
- Lai, Demirag, Leung – Tabu search for a vehicle routing
- Silvestrin, Ritt – Tabu search for a vehicle routing
- Soto et al. – Tabu search for a vehicle routing
- Lin, Bian, Liu – a Tabu search – simulated annealing hybrid with a dynamic neighborhood selection for TSP