# Object Oriented Programming 2018/19

Travelling salesmen problem by ant colony optimization

MEEC – IST

## 1 Problem

The *traveling salesman problem* (TSP) is one of the most important optimization problems since it belongs to a class of so-called NP-complete problems. Presently it is not known how to solve efficiently (in polynomial time) any NP-complete problem; therefore, approximations to the optimal solution are required. Ant colony optimization is a possible approach in this respect. The TSP has several applications, such as planning, logistics and the manufacture of microchips.

The TSP consists in finding the shortest cycle in a weighted graph that passes through all its nodes (only once). A *weighted graph* is a triple $G = (N, E, \mu)$ composed by a finite set of nodes $N$, a set of edges $E$, where an edge is represented by a set of two nodes, and a function $\mu : E \to \mathbb{R}^+$ that weighs each edge. As an example, consider the weighted graph depicted in Figure 1.

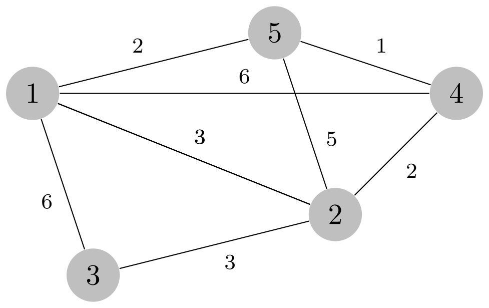

Figure 1: Example of a weighted graph.

A node is said to be *adjacent* to another node if there is an edge that connects them. A *path* is a sequence of nodes $n_1, \ldots, n_k$ such that, for every $i = 1, \ldots, k-1$, node $n_i$ is adjacent to node $n_{i+1}$. The weight of a path is the sum of the weights of the edges connecting consecutive nodes, that is,

$$\mu(n_1, \ldots, n_k) = \sum_{i=1}^{k-1} \mu(\{n_i, n_{i+1}\}).$$

Finally, a *cycle* is a path $n_1, \ldots, n_k$ such that $n_1 = n_k$. A cycle is said to be *Hamiltonian* if it contains all nodes and they occur only once in the cycle, except for $n_1$ which occurs twice. The TSP aims at finding the shortest Hamiltonian cycle in a weighted graph.

## 2 Ant colony optimization algorithms

The algorithms based on *ant colony optimization* (ACO) appeared recently for approaching difficult optimization problems on graphs. The idea is to simulate ants that randomly traverse the graph and that upon finding a solution to the problem lay down pheromone trails. The wide variety of algorithms, either for optimization or other hard problems, seeking self-organization in biological systems has led to the concept of *swarm intelligence*, which is a very general framework in which ant colony algorithms fit.

In the case of the TSP the ACO algorithm is governed by the following rules:

1. The level of pheromones of each edge is initialized to zero units.

2. Ants do not visit twice the same node, with the exception of the starting node, also called the *nest*. Therefore, an ant needs to store the path it traversed so far.

3. Let $i$ be the current node of the ant and $J$ the set of non-visited nodes adjacent to $i$. If $J$ is not empty, the ant randomly chooses one node $j$ in $J$ with probability proportional to the pheromone level of the edge connecting $i$ to $j$, and inversely proportional to the weight of this edge. Thus, if the ant is moving from node $i$ and it can move to nodes $J = \{j_1, \dots, j_\ell\}$ the probability of moving to $j_k \in J$ is given by

   $$P_{ij_k} = \frac{c_{ij_k}}{c_i},$$

   where:

   - $c_{ij_k} = \frac{\alpha + f_{ij_k}}{\beta + a_{ij_k}}$,
   - $c_i = \sum_{k=1}^{\ell} c_{ij_k}$,
   - $f_{ij_k}$ is the pheromone level of the edge connecting $i$ to $j_k$,
   - $a_{ij_k}$ is the weight of the edge connecting $i$ to $j_k$, and
   - $\alpha$ and $\beta$ are input parameters.

   Upon moving, the ant should update consistently its path.

4. If an ant has no other choice but to visit a node that was already visited ($J = \emptyset$) then it chooses randomly any adjacent node with a uniform distribution. Afterwards, it should update its path by removing the cycle created in the last move. As an example consider the graph in Figure 1. If the ant contains the path $1 \to 2 \to 3$ then it can only visit nodes 1 or 2, both already visited. One of these nodes should be picked up evenly; suppose the ant chooses to visit node 2. The path of the ant should remove the cycle $2 \to 3 \to 2$, and thus update its path to $1 \to 2$.

5. The time an ant takes to traverse an edge between nodes $n_i$ and $n_j$ has an exponential distribution with a mean value proportional to the weight of the edge, that is, with mean value $\delta \times a_{ij}$ where $\delta$ is an input parameter.

6. Upon completing a Hamiltonian cycle, and only in this case, the ant lays down pheromones in all edges constituting the cycle. The level of pheromones is inversely proportional to the weight of the cycle. In detail, the ant increments the level of pheromones in the edges of the cycle by

   $$\frac{\gamma W}{\mu(n_1, \dots, n_k, n_1)}$$

   units where:

- $W = \sum_{e \in E} \mu(e) = \sum_{i,j} a_{ij}$ and
- $\gamma$ is an input parameter.

After finding the Hamiltonian cycle, the ant restarts traversing the graph from the nest in order to find another Hamiltonian cycle.

7. The pheromone trail evaporates over time, thus reducing its attractive strength. This is performed by successive decreases in the pheromone level in the edges of the graph. The pheromones of each edge evaporate independently of each other and this evaporation occurs in discrete steps; in each of these steps, the pheromone level of the edge is reduced by $\rho$ units. The time between evaporations has an exponential distribution with mean value $\eta$, where $\eta$ is an input parameter.

## 3 Approach

The aim of this project is to give a solution in UML and Java to the TSP using the above ACO algorithm.

The simulation consists of generating at time zero an ant colony with $\nu$ ants and simulate this colony until the final instant $\tau$. It is assumed that the node $n_1$ representing the nest is an input parameter.

During the simulation, an ant will traverse the graph to find a Hamiltonian cycle and return to the nest, as described in Section 2. The shortest Hamiltonian cycle discovered until the current time of the simulation should be stored in order to be provided to the user whenever needed (see Section 5).

Ants traverse is ruled by **discrete stochastic simulation**, that is, based on a pending event container. The simulation is guided through the following discrete events:

- **Ant move**: An ant randomly chooses which node is going to move according to the rules 2-4 in Section 2. The time to traverse the edge from node $i$ to node $j$ has an exponential distribution with mean value $\delta \times a_{ij}$ (according to rule 5 in Section 2).
- **Evaporation of pheromone**: All edges with a positive level of pheromones trigger the respective evaporation event. The level of pheromones is reduced by $\rho$ units with an exponential distribution with mean $\eta$ between evaporations (according to rule 7 in Section 2).

The simulation ends when the next event to simulate occurs after the final instant $\tau$.

## 4 Parameters and results

The program should receive the following parameters:

| | |
|---|---|
| $n$ | number of nodes in the graph |
| $n_1$ | the nest node |
| $a_{ij}$ | weigh of traversing from node $i$ to node $j$ |
| $\alpha, \beta, \delta$ | parameters concerning the ant move event |
| $\eta, \rho$ | parameters concerning the pheromone evaporation event |
| $\gamma$ | parameter concerning pheromone level |
| $\nu$ | ant colony size |
| $\tau$ | final instant |

## 4.1 Input file format

The file that describes the input of the simulation is a XML file. The simulation is an element `simulation` whose attributes indicate the final instant $\tau$ (`finalinst`), the ant colony size $\nu$ (`antcolsize`) and the parameter $\gamma$ concerning the pheromone level (`plevel`). The element `simulation` contains only two other elements:

- The element `graph` whose attributes indicate the number of nodes $n$ in the graph (`nbnodes`) and the nest node $n_1$ (`nestnode`), contains the element `node`. The element `node` whose attribute indicates the index of a node (`nodeidx`), contains a list of elements `weight`. Each element `weight` has an attribute `targetnode` that identifies the index $j$ of the target node, and which content describes the weight $a_{ij}$ of traversing from node $i$ (stored in `nodeidx`) to node $j$ (stored in `targetnode`).
- The element `events` contain two new empty elements:
  - The element `move` whose attributes `alpha`, `beta` and `delta`, describe the parameters $\alpha$, $\beta$ and $\delta$, respectively, related to the move event.
  - The element `evaporation` whose attributes `eta` and `rho` describe the parameters $\eta$ and $\rho$, respectively, related to the evaporation event.

## 4.2 Example

Consider the graph depicted in Figure 1. A possible XML configuration follows:

```
<simulation finalinst="300.0" antcolsize="200" plevel="0.5">
    <graph nbnodes="5" nestnode="1">
        <node nodeidx="1">
            <weight targetnode="2">3</weight>
            <weight targetnode="3">6</weight>
            <weight targetnode="4">6</weight>
            <weight targetnode="5">2</weight>
        </node>
        <node nodeidx="2">
            <weight targetnode="3">3</weight>
            <weight targetnode="4">2</weight>
            <weight targetnode="5">5</weight>
        </node>
        <node nodeidx="3">
        </node>
        <node nodeidx="4">
            <weight targetnode="5">1</weight>
        </node>
    </graph>
    <events>
        <move alpha="1.0" beta="1.0" delta="0.2"/>
        <evaporation eta="2.0" rho="10.0"/>
    </events>
</simulation>
```

## 5 Results

During the simulation, the program should print to the terminal the result of observations of the system, realized from $\tau/20$ by $\tau/20$ time units. Each observation should include the present instant (*instant*), the number of move and evaporation events already realized (*mevents* and *eevents*), the Hamiltonian cycle (if there is any) found so far (*cycle*), according to the following format:

```
Observation number:
                    Present instant:                instant
                    Number of move events:          mevents
                    Number of evaporation events:   eevents
                    Hamiltonian cycle:              cycle
```

The observation *number* is given by $\tau/20$, and in total there are 20 observations in which the first observation corresponds to the instant $\tau/20$ and the last observation corresponds to the time $\tau$, i.e., the last observation corresponds to the end of the simulation.

The *cycle* should be printed, for the example in Section 2, as follows:

```
{1,5,4,2,3}
```

Any other printing to the terminal, or a print of this content out of this format, incurs in a penalty in the project grade.

## 6 Simulation

The simulator should execute the following steps:

1. Read the file that describes the input parameters of the simulation, and save/create the needed values/objects. The file with the input data is an XML file, whose format is described in Section 4.1, and it must be validated by an appropriate DTD.
2. Run the simulation loop until the final instant $\tau$ is reached (see Section 3). During the simulation, it must be printed to the terminal the population observations described in Section 5.

## 7 Evaluation

The assessment will be based on the following 7-point scale:

1. **(1.5 point)**: UML. The UML will be evaluated, in a 1.5-point scale as: 0–very bad, 0.4–bad, 0.8–average, 1.2–good and 1.5–excellent.
2. **(5.5 points)**: A Java solution that provides an extensible and reusable framework. The implementation of the requested features in Java is also an important evaluation criteria. The following is pre-established on a 5.5-point scale:
   (a) **(3.05 points):** Correct implementation of the simulation cycle, PEC, events, etc.
   (b) **(0.35 points):** Interfaces and polymorphism used correctly.

(c) **(0.35 points):** Open-closed principle used correctly.

(d) **(0.35 points):** `Object` class used correctly.

(e) **(0.35 points):** Correct use of data structures (`Collection`, etc).

(f) **(0.35 points):** Complete documentation of the simulator generated by the `javadoc` tool.

(g) **(0.35 points):** Five rich examples of input file parameters.

(h) **(0.35 points):** Final report with a critical evaluation of the proposed simulator.

Finally, on a 7-point scale the following discounts apply:

1. **(-0.35 points):** Prints outside the format requested in Section 5.
2. **(-0.35 points):** A non-executable jar file, or a jar file without sources or with sources out of date. Problems in extracting/building a jar file, as well as compiling/running the executable in Java, both from the command line. Problems with Java versions; the Java executable should run properly in the laboratory PCs.
3. **(-0.35 points):** Files submitted outside of the required format (see Section 8).
4. Projects submitted after the established date will have the following penalty: for each day of delay, there will be a penalty of $2^{n-1}$ points of the grade, where $n$ is the number of days in delay. That is, reports submitted up to 1 day late will be penalized in $2^0 = 1$ points, incurring in a penalty of 0.35 points of the final grade; reports submitted up to 2 days late will be penalized in $2^1 = 2$ points, incurring in a penalty of 0.7 points of the final grade; and so on. Per day of delay we mean cycles of 24h from the day and hour specified for submission.

## 8 Deadlines and material for submission

The **deadline for submitting the project is Thursday May 9, before 18:00**. The submission is done via fenix, so ensure that you are registered in a project group.

The following files must be submitted:

1. An UML specification including classes and packages (as detailed as possible), in `.pdf` or `.jpg` format. Place the UML files inside a folder named UML.
2. An executable `.jar` (with the respective source files `.java`, compiled classes `.class`, and `MANIFEST.MF` correctly organized into directories).
3. Five examples of input file parameters (`test_i`, with `i`$= 1, \ldots, 5$) used to test the program. Place these examples inside a folder named TESTS.
4. Documentation (generated by the javadoc tool) of the application. Place the documentation inside a folder named JDOC.
5. A final report (up to 10 pages, in .pdf format) containing information that complements the documentation generated by the javadoc tool. Place the final report inside a folder named DOCS.

6. A self-assessment form (in `.pdf` format) that will be made available in due time in the course webpage. Place the self assessment form inside the folder named DOCS (the same folder as the final report).

The UML folder, executable (the `.jar` file with the source files, besides the compiled files and `MANIFEST.MF`), the JDOC folder and the DOCS folder, should be **submitted via fenix in a single `.zip` file**. Only .zip is allowed.

**The final discussion will be held from May 20 to May 30.** The distribution of the groups for final discussion will be available in due time. All group members must be present during the discussion. **The final grade of the project will depend on this discussion, and it will be not necessarily the same for all group members.**