Object Oriented Programming 2022/23

Travelling salesmen problem by ant colony optimization

LEEC/MEEC – IST

1 Problem

The Traveling Salesman Problem (TSP) is a crucial optimization problem belonging to a class of NP-complete problems. No known efficient (polynomial time) algorithm exists to solve any NP-complete problem. Therefore, approximations to the optimal solution are necessary. Ant colony optimization is a promising approach in this regard. The TSP has several practical applications, including planning, logistics, and microchip manufacturing.

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.

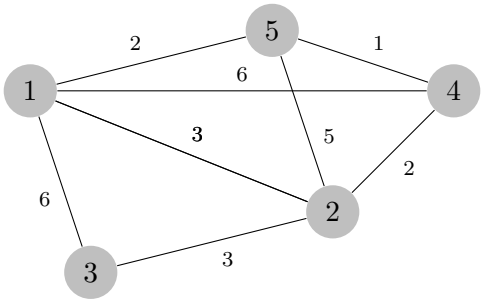

Figura 1: Example of a weighted graph.

A node is said to be *adjacent* to another node if an edge 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 to find the shortest Hamiltonian cycle in a weighted graph, that is, the path with the lowest weight.

2 Ant colony optimization algorithms

The algorithms based on *ant colony optimization* (ACO) appeared recently for approaching complex graph optimization problems. The idea is to simulate ants that randomly traverse the graph and lay down pheromone trails upon finding a solution to the problem. 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 following rules govern the ACO algorithm:

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

2. 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 $J = \{j_1, \ldots, j_\ell\}$ 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 connecting i to j_k, and
   - α and β are input parameters.

   Upon moving, the ant should update its path consistently.

3. If an ant has no choice but to visit a node already visited ($J = \emptyset$), it randomly chooses 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 $1 \to 2 \to 3$, 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 $1 \to 2$.

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

5. 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 at the edges of the cycle by

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

   units where:

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

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

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

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

3 Approach

This project aims to give a solution in UML and Java to the TSP using the above ACO algorithm.

The simulation consists of generating an ant colony with ν ants at time zero and simulating this colony until the final instant τ. 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 simulation's current time should be stored to be provided to the user whenever needed. The simulation is guided through Discrete Stochastic Simulation (DSS) with the following discrete events:

- Ant move: An ant randomly chooses which node will move according to rules 1-3 in Section 2. The time to traverse the edge from node i to node j has an exponential distribution with mean value $\delta \times a_{ij}$.
- Evaporation of pheromone: All edges with positive levels of pheromones trigger the respective evaporation event. The level of pheromones is reduced by ρ units with an exponential distribution with mean η between evaporations (according to rule 7 in Section 2).

The simulation ends when the next event to simulate occurs after the final instant τ.

4 Parameters

The program should run from a JAR file named `project.jar` and 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, for all $i, j \leq n$
α, β, δ	parameters concerning the ant move event
η, ρ	parameters concerning the pheromone evaporation event
γ	parameter concerning pheromone level
ν	ant colony size
τ	final instant

The program can be invoked from the command line in two different ways. The first way is:

`java -jar project.jar -r` $n\ a\ n_1\ \alpha\ \beta\ \delta\ \eta\ \rho\ \gamma\ \nu\ \tau$

This command does not contain a graph. Therefore, a random graph with a Hamiltonian cycle must be generated, with the specified number of nodes n and a maximum edge weight a. The minimum edge weight should be zero. Keep in mind that although the generator must ensure a Hamiltonian cycle, the resulting graph may also contain other cycles. The input parameters (n, n_1, α, β, δ, η, ρ, γ, ν, and τ), alongside the generated graph, should be used to run the simulation.

The second way to invoke the program looks like this:

`java -jar project.jar -f` $< infile >$

where $< infile >$ is a text file (`.txt`) with all parameters needed for simulation, including the graph. The graph should be displayed as an adjacency matrix that shows the weight of each edge if an edge exists in the graph, and zero otherwise. The format of $< infile >$ is as follows:

n n_1	α β	δ η	ρ γ	ν τ
0	a_{12}	a_{13}	...	a_{1n}
a_{21}	0	a_{23}	...	a_{2n}
...	...	...	...	...
a_{n1}	a_{n2}	a_{n3}	...	0

where $a_{ij} = a_{ji}$ and $a_{ii} = 0$, for all $i, j \leq n$.

Therefore, for the example in Figure 1, we have an `input.txt` file on disk representing the simulation with:

```
5    1    1.0  1.0  0.2  2.0  10.0  0.5  200  300.0
0    3    6    6    2
3    0    3    2    5
6    3    0    0    0
6    2    0    0    1
2    5    0    1    0
```

During the parsing of the `input.txt` file, spaces and tabs should be ignored. If any other errors occur, the program should be aborted, and an explanation should be given to the user. Moreover, the `input.txt` file can be invoked from any location on the disk, using either a fixed or relative path with respect to the executable. In the previous example, the `input.txt` file was located in the same directory as the executable. However, the invocation should use a relative path if a user wishes to have a `TESTS` folder containing multiple test scenarios alongside the executable. For example:

```
java -jar project.jar -f ./TESTS/input.txt
```

or just:

```
java -jar project.jar -f TESTS/input.txt
```

Avoid hardcoding the location of files/directories in your project to ensure it can run on any computer. Hardcoding the file/directory location will result in a penalty to your project grade.

5 Results

Before starting the simulation, the program should print the input parameters to the terminal as follows:

Input parameters:

n	:	number of nodes in the graph
n_1	:	the nest node
α	:	alpha, ant move event
β	:	beta, ant move event
δ	:	delta, ant move event
η	:	eta, pheromone evaporation event
ρ	:	rho, pheromone evaporation event
γ	:	pheromone level
ν	:	ant colony size
τ	:	final instant

with graph:

$$\begin{matrix} 0 & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & 0 & a_{23} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & a_{n3} & \dots & 0 \end{matrix}$$

During the simulation, the program should print to the terminal the result of system observations, 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 shortest Hamiltonian cycle (if there is any) found so far (*best*), according to the following format:

Observation *number*:

Present instant:	*instant*
Number of move events:	*mevents*
Number of evaporation events:	*eevents*
Top candidate cycles:	*cycles*
Best Hamiltonian cycle:	*best*

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 τ, i.e., the last observation corresponds to the end of the simulation.

The *best* Hamiltonian cycle found so far should be printed, for the example in Section 2, as follows:

{1,5,4,2,3}:14

If such a cycle has not been found yet, just print to the terminal {}. However, at a certain stage in the simulation, ants diligently strive to discover alternative cycles. For the purpose of debugging, it is required to print the top (up to) five cycles discovered so far, referred to as *cycles*. These cycles should be distinct from the best cycle, if any, and should be printed in descending order of quality, with the best cycle appearing first. For example:

```
{1,3,2,4,5}:14
{1,3,2,5,4}:21
{1,4,5,2,3}:21
```

Note that both `cycles` and `best` are accompanied by their respective weights.

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

6 Simulation

The simulator should execute the following steps:

1. Read the input parameters of the simulation and save/create the needed values/objects.
2. Run the simulation loop until the final instant τ is reached (see Section 3). During the simulation, the population observations described in Section 5 must be printed to the terminal.

7 Examination and grading

7.1 Deadlines

The deadline for submitting the project on Fenix is (before) **June 16, 16:30**. The project accounts for 10 points of the final grade, which are distributed as follows:

1. **(2.5 points) UML**
   - The UML specification, including classes and packages (as detailed as possible), in `.pdf` format. Only `.pdf` format is accepted, with the file named `UML.pdf`.
2. **(7.5 points) Java**
   - The executable `project.jar` (with the respective source files `.java`, compiled classes `.class`). Source files are mandatory.
   - Javadoc documentation (generated by the Javadoc tool) of the application, placed inside a folder named `JDOC`.
   - At least five compelling examples with the corresponding input files and respective simulations. These examples should include something other than scenarios offered on the course webpage regarding this project and trivial variations thereof. The scenarios and their simulations should be placed in a folder named `SIM`, i.e., the `SIM` folder should contain the input files with the scenarios (for instance, `input1.txt`, `input2.txt`, etc) and the respective text files (`.txt`) with the results from the simulation (for instance, `simscenario1.txt`, `simscenario2.txt`, etc).

   The `UML.pdf`, the executable `project.jar`, the Javadoc documentation and at least five input/simulation examples, should be submitted via fenix in a single file before **June 16, 16:30**. (the `.pdf` and `.jar` files and the folders `JDOC` and `SIM` should be submitted in a single `.zip` file). Only files with `.zip` extension are accepted. A self-assessment form must also be submitted via Forms in the Teams (before the end of May 16).

3. **Final discussion: 20-23 June 2023** The distribution of groups for the final discussion will be available at a later time. All group members must be present during the discussion. The final grade of the project will depend on this discussion, and it may not be the same for all group members. Regardless of the IDE used during project development, all students should be proficient in using Java on the command line.

7.2 Assessment

The assessment will be based on the following scale of cumulative functionality, where each level corresponds to a maximum grade. On a 10-point scale:

1. **(2.5 points)**: UML solution (0 points – non-existing, 0.5 points – bad, 1 point – sufficient, 1.5 points – good, 2 points – very good, and 2.5 points – excelllent).
2. **(7.5 points)**: Java implementation.
   (a) input, randomly generated graphs with Hamiltonian cycle (1 point)
   (b) simulation (4.5 points)
   (c) output, finding the best Hamiltonian cycle (0.5 points)
   (d) Javadoc documentation (1 point)
   (e) examples of input/simulation (0.5 points)

The implementation of the requested features in Java, specific project requirements, and the quality of the oral discussion, are also important evaluation criteria, and the following discounts (on a 10-point scale) are pre-established:

1. **(-2 points):** OOP ingredients are not used or they are used incorrectly; this includes polymorphism, open-close principle, etc.
2. **(-1.5 points):** Java features are handled incorrectly; this includes incorrect manipulation of methods from `Object`, `Collection`, etc.
3. **(-1 points):** A non-executable JAR file or a JAR file without sources.
4. **(-0.5 points):** A submitted .zip outside of the requested format.
5. **(-0.5 points):** Hardcoded input file (as explained in Section 4).
6. **(-0.5 points):** Prints outside the format requested in Section 5.
7. **(-1.5 points):** Individual assessment of the student participation in the oral project discussion (on a per-student basis, rather than as a group).
8. **(-1.5 points):** Individual assessment of student ability to extract/build a JAR file, as well as compile/run the executable in Java from the command line (on a per-student basis, rather than as a group).

Projects submitted after the established deadline will incur a penalty. For each day of delay, a penalty of 2^{n-1} points will be deducted from the grade (on a 10-point scale), where n is the number of days delayed. For instance, projects submitted one day late will be penalized by $2^0 = 1$ point. Projects submitted two days late will be penalized by $2^1 = 2$ points, and so on. A day of delay is defined as a cycle of 24 hours from the specified submission date.