INSTITUTO SUPERIOR TCNICO

OBJECT ORIENTED PROGRAMMING

Travelling salesmen problem by ant colony optimization

Authors:

Francisco Costa 81673

Pedro Mendes 81046

Rui Livramento 81051

Group: 9

Professor

Alexandra Carvalho

May 9, 2019

Contents

1 Introduction

The main goal of this project was to develop a simulator to solve the *traveling salesman problem* (TSP), i.e., find the best path in a weight graph that passes through all nodes only once and, in the end, returns to the first node. To solve this problem, an ant colony optimization (ACO) algorithm was implemented and it was used a stochastic simulation based on events that act upon a population and on the graph.

In the specific case of this project, a population of ants was used which can traverse between the nodes of the graph in order to try to solve the TSP. How the simulation evolves is determined by move and evaporation events that, in this case, happen over the ants of the colony and the edges of the graph respectively.

In the next sections will be described the approach that was taken to solve the problem, how the solution was structured and organized, and why some decisions were made, as well as an overall evaluation of the obtained solution, using several different tests, each testing a specific feature of the developed work.

2 Packages

One of the goals during the development of this project was to implement a solution for the traveling salesman problem (TSP) that could be generalized as must as possible. A correct division of the project had to be done in order to achieve this goal and to be possible a re-utilization of the code for different types of simulation (stochastic and deterministic simulation), or for different events or populations.

To achieve this objective, there was a division of the project by packages, one for the simulation, one for the events, one for the graph and the last one for the colony of ants and its simulation. This division helps to achieve the previous goal, in the way that all classes and methods that have a relationship are implemented in the same packages. Also, the classes that can be generalized are implemented in a different package from those that are specific to the problem purpose (that are implemented in `colony` package of 2.3). The packages that are generalized can be used independently or can be easily substituted by other and new functionalities.

2.1 simulation

In order to fulfill the Open-close principle, the simulation is implemented in a unique package that contains all the classes responsible for the stochastic simulation based on events. This package is composed of three different classes.

The `Simulation` has a Pending Event Container (PEC) that stores the events by an order to be executed in a given time. The `Simulation` also has two attributes, one to store the maximum simulation time and another one to store the current simulation time, in order to track the current time of the events. This class is responsible to get the next event in the PEC and triggers it. It also verifies the end of simulations, i.e., when the PEC is empty, because an event is only added to the PEC if the time instant to trigger the event is smaller than the maximum simulation time.

The simulation runs step by step and, in each step, triggers events in temporal order, that is, the first event in the PEC. The current time of simulation is updated with the event time that was triggered. Events that have the trigger time higher than the simulation time are never added to the PEC.

The `PEC` implements a data structure to store the events. In this particular case, it's used a priority queue that sorts the events by ascending order of the event time. A priority queue was chosen because it provides an easy way to, orderly, add and remove events from the data structure.

2.2 events

The events are inherited (extended) from the class `Events` implemented in the package `simulation` and they are implemented in a separated package in order to facilitate the addition of different events and also to provide an easy way to reuse these events to different populations, without being restricted to the one used in this project. For example, the population used in the simulation can be an ants' colony or a bees' colony, and the population will have different behaviors depending on the algorithms used. These different behaviors can correspond to new, different and unique events that depend on the population simulated, which can be created in this package. Also, there might be other events as for example, the death or reproduction of an individual.

In the particular case of this report, three different events were implemented, the `Move`, `Evaporation` and `Observation` events.

Each class has an abstract function that is implemented in the package related to the population because a certain event can manage the procedure differently depending on the population and the map that is being simulated.

The classes of this package provide a polymorphism to the project. As seen before, it's possible to have any type of events that share the inheritance of the class `Events`.

2.3 colony

The population is created in a package `colony` and implements all the classes and the respective methods to handle the colony of ants. The simulation of an ant's colony is created to extend the class `Simulation` of 2.1. This simulation starts a colony that creates the ants and saves the object to the graph.

The methods to handle the graph are defined using the `GraphInterface` interface, in order to achieve "complete abstract level", so the colony does not have any access to the implementation of methods related with the graph, i.e., the graph is completely independent of the simulation of the colony and vice versa.

The `SimulationColony` class has two objects in order to handle the `Move` and `Evaporation` events. There are also three additional classes that implement the abstract functions define in 2.2, because the way to manage the event is dependent on the type of population and map simulated.

In the implementation of this package was hard to extend the code and use it in a different problem to ensure the open-closed principle. This package has objects that extends the `Simulation` of 2.1 and the events of 2.2. In order to achieve this goal, a generic type population could be created (instead of restringing to a population of ants). However, as seen before, for different populations, the same events can have different impacts on the population and on the map. So, different methods should be created for a different population. During this project, the only focus was on ants' population in order to solve the proposed problem.

2.4 graph

To solve the TSP a map has to be given and, in this case, is represented by a graph. This package contains all the information and methods to handle the graph that is implemented using an adjacency list.

This package has three classes. The `Graph` that saves the information given in the input file (for example, the nest node) and also has a vector of `Nodes` of size equal to the number of nodes in graphs. Each `Node` has a `LinkedList` of `Weight` to store the neighbors' identifiers, and the correspondent weight and pheromone level of an edge.

In the class `Graph` the methods defined in the interface of 2.3 are implemented.

2.5 tsp

The `Main` class was implemented in this package, where it reads and interprets, using the `SAXParser` Class, the input XML file. The `SAXParser` Class was used because it gives an easy way to read and parse the information of the input file. Using the information read, the `graph` and `simulation` objects are created and the simulation of the colony is run.

3 Tests

3.1 Map 1

The objective of this map was to test how the simulation performs on big maps, in this case, a graph with 75 nodes, each one with random number of neighbors and random weights on the edges between them. In order to test this, it was used a large simulation time, 100000, and a small colony size, 300 ants, to verify if even one Hamiltonian cycle was found. The results were as expected, the simulation found an Hamiltonian cycle after a few observations, showing that for small colonies, if given enough time it will find a solution, if there is one. However, if the size of the maps and/or the colony increases significantly, the problem will become increasingly more complex making it not feasible in useful time.

3.2 Map 2

Running a fully connected graph of 100 nodes, 4950 edges and with 10000 ants shows that our implementation of the schedule of events in the PEC is the most efficient one. As this is a very complex map not only in terms of a number of ants but also in the number of the nodes and edges that leads to a huge number of different paths, we wouldn't have enough computational power and memory to schedule all of the events at the same time. Instead, we only schedule the next event of an ant or of an edge when the previous one ends. With this implementation, the maximum number of events in the PEC, in any given time, is the total number of edges (as each one can have a corresponding evaporation event) plus the number of ants (as each one can have a corresponding move event) plus one, which corresponds to the next observation event. The maximum number of events in the PEC is then given by,

$$\max(\#events) = \#edges + \#ants + 1. \tag{1}$$

3.3 Map 3

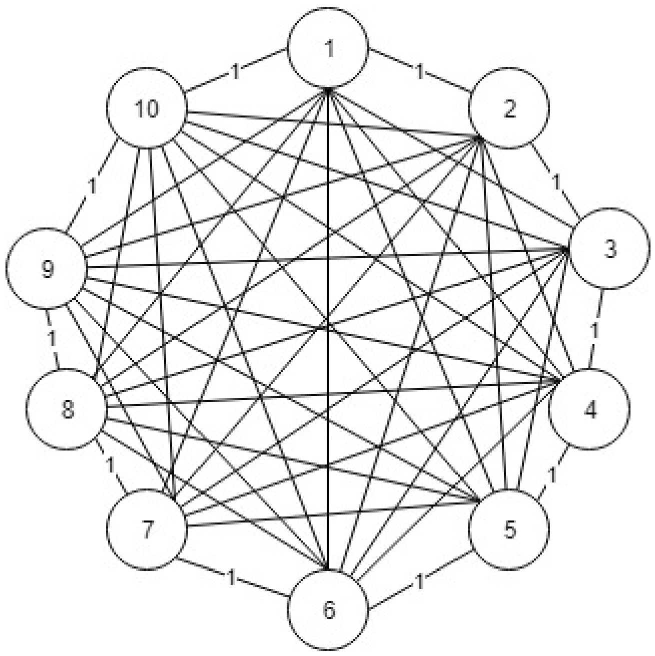

Figure 1 – Fully connected graph

In this map, we try to evaluate the influence of the pheromones and evaporation. In this fully connected graph with 10 nodes, there are many Hamilton cycles but only one minimal path with a total weight of 10 which corresponds to the outside circle connecting all nodes, as we can see in figure (1).

3.3.1 High evaporation rate

The simulation, for a high evaporation rate, i.e, a small time between evaporation events and/or a high decrease value, is constantly finding new, better Hamiltonian cycles instead of improving the previous path, leading, most of the times, to slow convergence. For a η=0.5 and ρ=100 the simulation will take too much time to find the optimal path as will be randomly choosing new paths without the influence of the pheromones.

3.3.2 Low evaporation rate

On other hand, if we have a low evaporation rate, i.e, a big time between evaporation events and/or a low decrease value, the simulation, once it finds a Hamilton cycle, will have a difficult time finding better routes because the ants will have an accessibly preferable path that will have an increased probability of being chosen. For this map, a η= 100 and ρ= 0.5 leads to a ”stubborn” simulation which will, for most of the times, be ”stuck” in a non-optimal Hamiltonian cycle for a long period before finding the optimal one.

3.3.3 Optimal evaporation rate

If we consider these two extremes, it becomes clear that an optimal, or at least a semi-optimal combination of the time between evaporation events and the decrease rate must be found in order to ensure, in most of the times, a fast convergence. If we use a η=2 and ρ=10 the optimal path is usually found within very few observations

3.4 Map 4

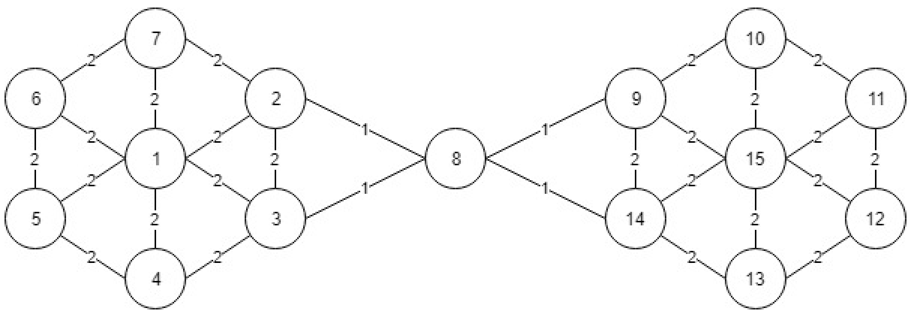
Figure 2 – Map with no solution

In this map there is no Hamilton cycle as the two clusters are only connected via one node, making it impossible for an ant to come back to the nest node without repeating the central node. As predicted, our simulation cannot find a solution no matter the number of ants and time. In terms of optimization, to better solve these types of graphs a check before the simulation itself could be implemented as they can be recognizable.

3.5 Map 5

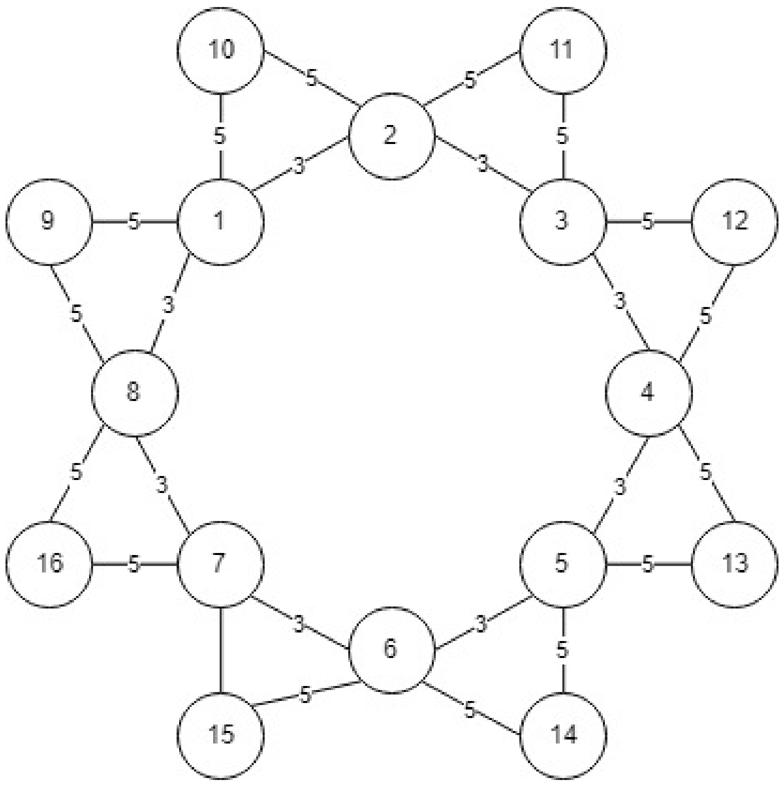

Figure 3 – Map with only 1 possible Hamiltonian cycle

In this map, it is evaluated how hard is for the algorithm to find a Hamiltonian cycle when there are a high number of different path to chose but there is only one possible Hamiltonian cycle. To add up to this difficulty, the weights of the edges were changed so that the probability of finding the right path is smaller. This was achieved by giving higher weights to edges connecting to the outer nodes and lower ones to the inner nodes edges, as we can see in figure 3. As expected, it often takes some observations until it finds an Hamiltonian cycle as the simulation will, for most of the trials, be forced to reset some of its path when it has no choice but to repeat nodes.

4 Conclusion

One of the main concerns during the development and design of this project was to respect the Open-closed principle and implement a solution that, using the features of Java, is easily extensible to other problems.

However, as seen before, sometimes it's hard to program avoiding the creation of dependencies between packages. This happened in the `colony` package, that extends the `simulation` and the `events` to be possible to simulate a specific population of ants. This fact creates a barrier that makes the program less prone to extension to other purposes.

To try to attenuate this fact and to facilitate the re-utilization of the code, at least one interface for each package could be implemented, in order to avoid the direct dependencies and inheritances. However, in most of the cases, it's troublesome to determine exactly where and how an interface could be implemented. Another improvement could be the implementation of a generic population, instead of being restricted to the Ant Colony Optimization.

However, all the remaining packages and classes were implemented without creating strong dependencies and inheritances between packages, in order to try to guarantee the code re-utilization in different problems, the easy substitution for other implementations and the introduction of new features of the TSP using ant colony optimization in stochastic simulation.

To sum up, as analyzed in 3, the implemented algorithm solves the TSP, however, it has some restrictions that can lead to not achieving a solution, or a non-feasible run, depending on the graph complexity and the colony's size. To try to mitigate these restrictions, there could be an optimization on some parts of the problem, nonetheless, this is not covered in the purpose of this project.