Instituto Superior Técnico

Object Oriented Programming

MEEC

Travelling Salesmen Problem by Ant Colony Optimization

Group 33
Francisco Melo - 84053
Rodrigo Rego - 89213
João Guedes - 78338

May 9, 2019

1 Introduction

The *travelling salesman problem* (TSP) consists in finding the shortest route that passes through every specified point once and returns to a given origin point, where the distance is defined as a cost between points.

The TSP can be modelled as an undirected graph and formulated as an optimization problem - starting and finishing at a specified node after having visited each other node exactly once and minimizing the overall cycle's distance.

Goal

The goal of this project is to use a kind of *swarm intelligence* framework to solve the minimization problem - *ant colony optimization* (ACO).

The idea is to simulate ants that randomly traverse the graph and that upon finding a solution to the problem lay down pheromone trails that evaporate over time.

The ACO approach provides a very intuitive perspective to understand how can it achieve a solution in an optimal way - the pheromone trails will enable a less random choice of traverse upon finding a solution, and their evaporation over time will counteract convergence to local optimal solutions.

To implement the problem and simulate the ant colony, graph theory and an OOP approach using Java are used.

2 Data Structures

Different data structures are implemented in different parts of the problem in order to serve a specific need or in order to gain some efficiency.

Data structures are important to implement: the **graph**, the **pending event container (PEC)** which is a major piece of the discrete stochastic simulation approach, the gradually updated **traversed path of each ant** and the **ant colony**.

Each of these entities is implemented as a class with an attribute implementing the chosen data structure.

Graph

- **Data Structure:** `ArrayList`

The graph is implemented as an `ArrayList` of Nodes and each Node is consecutively an `ArrayList` containing Edges - node's neighbours and the cost associated to traversing each neighbour.

Why?

- The graph has to be accessed many times throughout the simulation time, so accessing its elements very efficiently can be achieved with an `ArrayList`.
- Accessing: $\mathcal{O}(1)$.
- The size of the graph (length of the `ArrayList`) varies.

Pending Event Container (PEC)

- **Data Structure:** Sorted `PriorityQueue`

The PEC is implemented as a sorted `PriorityQueue` of Events, where sorting is accomplished with a Comparator which compares by the event's time value.

Why?

- The PEC must return the event with the smallest time value many times throughout the simulation, so an efficient way of doing this is by using a sorted structure that easily returns and removes the event that should be simulated first. This can be accomplished with a sorted `PriorityQueue` with a Comparator.
- The PEC must also store an unknown number of elements.

Ant's Traversed Path

- **Data Structure:** `LinkedList`

The ant's traversed path is implemented as a `LinkedList` of integers which stores the number of the nodes the ant has traversed so far.

Why?

- It is important to add/remove elements in arbitrary positions of the list in order to gradually build the traversed path (especially when the ant decides to visit a previous node, all nodes until the duplicate's original must be removed). It is known that a `LinkedList` is better for this kind of application.
- The length of the list varies.
- Accessing and Add/Remove (position i): $\mathcal{O}(min\{i, n-i\})$, where n is the length of the list (less efficient access compared with an `ArrayList`).

Ant Colony

- **Data Structure:** `ArrayList`

The ant colony stores all the ants used during the simulation.

Why?

- Efficient accesses are valued in this application.

3 OO Solution

Given the explained goal, there should be a concern and an emphasis on trying to make the code extensible.

The simulation of ACO algorithm is done through a discrete stochastic approach which can naturally be extended to many other simulation related applications

⇓

Simulation related classes should be implemented in a separate package: `simulator`

⇓

Use of Interfaces and Polymorphism should be maximized

⇓

– Interfaces –

- `simulator.PendingEventContainer`
- `simulator.IEvent`

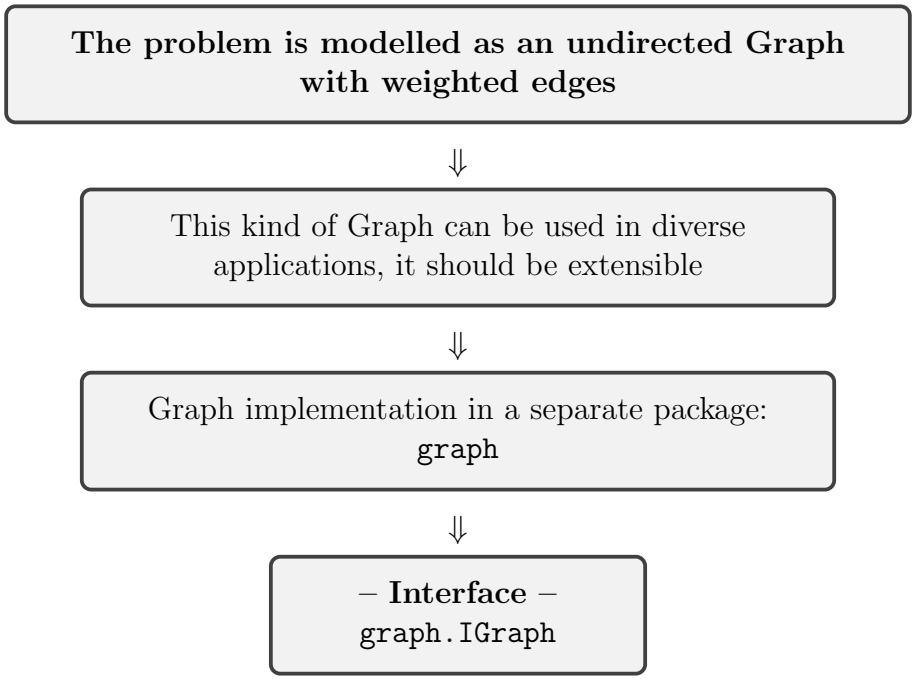
– Abstract Classes –

- `simulator.Simulator`
- `simulator.Event` (Event implements IEvent)

- **Interface** `simulator.PendingEventContainer`: this interface allows the implementation of indispensable methods necessary to build the PEC.
- **Interface** `simulator.IEvent`: this interface allows the implementation of classes that can be used to create event objects that can function with the PEC.

- **Abstract class** `simulator.Simulator`: allows the `Simulator` to be extended to different types of simulations (in this case an ant colony) which ultimately require and share a particular set of attributes.
- **Abstract class** `simulator.Event`: implements the interface `IEvent` and allows the `Event` to be extended to other classes that share some attributes and that represent a particular kind of event (`AntMove` and `PheromoneEvaporation` are examples of events).

This particular implementation of the simulation related classes is one solution of how to generalize the package `simulator`, thus considering extensibility.



- **Interface** `graph.IGraph`: allows the implementation of a graph class with methods of general usage associated to graph manipulation.

These two packages were structured this way in order to achieve the extensibility of the code.

The classes that specialize in the simulation of the ant colony itself - for instance, the events `AntMove`, `PheromoneEvaporation`, classes to read XML test files, the simulation class with the main method `AntSimulation`, among others, - are implemented in a different package `antColony`.

In addition to this, however, the extensibility of the code is also related to the **visibility** associated to each class, method and attribute.

Visibility - `simulator`

- `Event`

  - *Public* class with only a constructor method.

  - `time` attribute is set to *protected* so it can be accessible to the event subclasses in the `antColony` package.

  - The comparator class is set to *package*.

- `PEC`

  - *Public* class with *public* methods.

  - `PriorityQueue<Event> events` attribute is set to *private*, in order to be protected from everywhere else but the class itself. The pending events should be protected in this way to avoid simulation tampering.

- `Simulator`

  - *Public* class.

  - All its attributes are set to *protected*, as they are simulation's parameters that should be protected from the world.

These visibility choices work with the structural decisions defined before, together they allow the package extensibility.

Visibility - `graph`

- `Graph`, `Node` and `Edge`

  - To allow a general implementation of the graph in other applications all classes, methods and attributes are *public* except the `Node.ArrayList<Edge> edges` that is set to *package* visibility in order to be only visible from within the graph package (it isn't necessary to be visible elsewhere).

These visibility choices work with the structural decisions defined before, together they allow the package extensibility.

Visibility - `antColony`

- Visibility summary (general view)

  - The extended event classes are set to *package*.

  - Static attributes of the type *Random* necessary to compute exponential distributions are used in *private* methods.

  - The `Ant` class is *package* and the structure `LinkedList<Integer> traversedPath` is set to *package*.

  - Classes, methods and attributes related to reading XML test files are *public*.

  - The `AntSimulation` class is *public* with simulation related attributes set to *package*.

The visibility choices made in this package were not specifically intended to consider the extensibility of the code since it is specifically related to the ant colony simulation itself.

This package is, in some way, a specialization of the more generally coded concepts of the other two packages.

4 Application Performance

The application's performance depends, in a first analysis, on the correct choice of data structures. It is believed that the choices made regarding this issue provide a good level of confidence regarding efficiency and the correct implementation of the described problem.

In one of the tests, a 500 node graph with an *hamiltonian* geometry was run by the application without issues. This shows that the application can handle big problems in very reasonable time.

On a critical note, we believe that the application should be better protected with a deeper study of exceptions and input error control at execution. Moreover, we suspect that the number of evaporations is a bit excessive, despite our debugging efforts we couldn't manage to figure it out on time. However we checked that pheromone evaporations definitely occur throughout the simulation time and accordingly to the ρ value.

5 Tests

5.1 Test 1

`test_1.xml`

This test is the one given in the course web page. The purpose of this test was to check if it was working properly for a simple graph. For this test the shortest Hamiltonian cycles found were: {1, 5, 4, 2, 3 } and {1, 3, 2, 4, 5 }. Figure 1 shows the graph of this test.

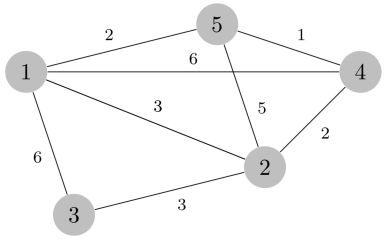


Figure 1: Graph considered in Test 1

5.2 Test 2

`test_2.xml`

For this test we consider a node without edges (node 5), that means that it is impossible to have a Hamiltonian Cycle. The purpose of this test is to check the response of the program when it is impossible to find a Hamiltonian Cycle. Figure 2 shows the graph of this test.

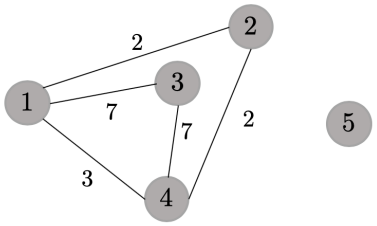


Figure 2: Graph considered in Test 2

5.3 Test 3

`test_3.xml`

The purpose of this test is to check the response of the program to a graph with multiples Hamiltonian cycles with the same weight. For this test the shortest Hamiltonian cycles found were: {1, 3, 7, 5, 6, 8, 4, 2 }, {1, 5, 7, 3, 4, 8, 6, 2 } and {1, 2, 4, 8, 6, 5, 7, 3 }.
Figure 3 shows the graph of this test.

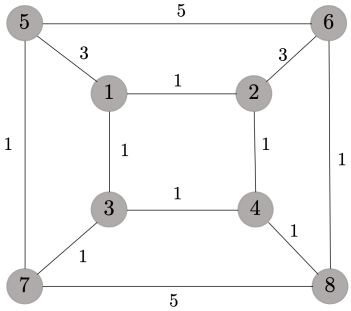


Figure 3: Graph considered in Test 2

5.4 Test 4

`test_4.xml`

This test provides a graph with 500 nodes all connected. The purpose of this test was to check the performance of the program.

5.5 Test 5

`test_5.xml`

To check the effect of the parameters related to the number evaporation, ρ was changed. This change was made in the graph of the Test 1. Decreasing the value of ρ the number of evaporation's reduced in comparison with the Test 1.

6 Conclusions

The goal of this project was to apply ACO algorithm in order to solve the TSP through an OOP implementation.

Given the results and tests performed it can be concluded that the application performed well and could solve the problem even for a graph of 500 nodes. Moreover, the number of movement events and the hamiltonian cycles given as solutions agreed with our intuition and with what was expected. However, despite our debugging efforts, we think that the number of evaporations is not entirely correct in some cases (a bit excessive), still evaporations definitely occur in accordance to the ρ value.

Finally, the time it took to converge to a solution was always acceptable and the randomness was checked for an example in which we had at least two smallest cycles with the same cost - it would converge roughly as much to one of the cycles as to the others.