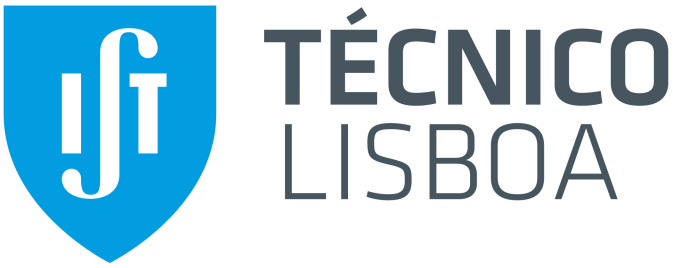

Instituto Superior Técnico

Master in Electrical and Computer Engineering

Object Oriented Programming

The Travelling Salesman Problem

Duarte Correia, duartecorreiascs@hotmail.com
João Pinto, joao.s.pinto@tecnico.ulisboa.pt
José Bastos, jose.antonio@tecnico.ulisboa.pt

Número:
81225
84085
87787

Group #35

9th May 2019

1 Introduction

The main objective with this work was to implement an algorithm to the *Travelling Salesman Problem* (TSP), a so-called NP, *Non-Deterministic Polynomial time* problem meaning it is solvable in polynomial time by a non-deterministic *Turing machine*. For this we had to develop the code in Java, an object-oriented language.

The problem consists of a set of nodes connected with a specific weight between them and it is intended to find a solution for the best path (with the lowest weight) that contains every node only once.

For that it is used an approach called *ant colony optimisation* (ACO). In the ACO, a set of cooperating agents called ants cooperate to find good solutions. They cooperate by laying their pheromones in the path. What happens is that when an ant has to make a decision it chooses a random possible path according to the weight of the edges and the amount of pheromones in each path. If we have a population of ants big enough the pheromones will make most ants go across possibly not the best but at least a good path after a while. This means we can not obtain 100 percent sure that the optimal solution is given, but we can get a very good one without having to try all the cases.

The simulation is done with the help of a *Pending Event Container* (PEC). It works like a queue of the future events where a cycle iterates over the events waiting in the PEC to happen. These events can be an ant event where an ant moves from one node to another, an evaporation event where pheromones are lowered and observation event where some information is printed to the user interface.

2 Project objectives

For this work we defined some objectives for the development. First, we had to use the most important features of Java, such as,

- Organising the classes in *Packages* since we can change methods and variables visibility according to what is needed.
- Using *Interfaces* to achieve security hiding some details of objects and only show the important details.
- Taking advantage of *Polymorphism* to implement different methods for several classes that have similarities between them
- Using *API*'s to implement routines like parsing an XML file

And in terms of code it had to be:

- *Extensible* so it could be easier to take it to further developments since it is an object-oriented language.
- *Robust* so all the possible errors could be handled with user defined exceptions.
- *Clean* and *consistent* (indentation, variable names, etc.) so it could be understood easily by other people.

After the coding, the objective was to make several tests of the simulation of the ACO. For that we had to understand the initial parameters provided in the XML so we could change them and search for an optimal solution.

3 Main data structures

The core of the solution for the problem in hands, as of all software applications developed, is mostly related to the data structures chosen for keeping the information, relevant to that problem, to be accessed, organised and manipulated so as to solve it. In addition, another main aspect of selecting the correct data structure is the time complexity of the various operations performed in a program. For instance, if the values contained in some data structure need to be constantly accessed, it is not, in any way, desirable that obtaining those values takes infinitely many time, as the program would require an excessive amount time to provide a solution.

The problem of the *Travelling Salesman Problem* (TSP) is broken down into the following main segments: the weighted graph, for which the objective is to find the cheapest *Hamiltonian* cycle; the ants, which are responsible for finding that cycle; and, at the last, the *Pending Event Container* (PEC) that coordinates the sequence of events that influence how each ant moves through the various paths of the graph. For each of these segments, the data structure which supports each of them was chosen regarding their purpose, in spite of the program time efficiency, and memory usage.

3.1 Graph

A graph is, most often, implemented using an adjacency matrix or adjacency lists. Before making a decision on which data structure should be employed, one should think what is the purpose of the graph in this problem. One of the crucial aspects of the problem is the following, given that an ant is one a certain vertex and it needs to know to go next. Hence, access to all edges leading to all neighbouring vertices. Finally, it is convenient that the graph is light on memory.

For the reasons above mentioned, it is obvious that the adjacency lists implementation is more suitable for this situation. Creating the graph from scratch takes a time interval proportional to the number of vertices, $\mathcal{O}(V)$, returning all neighbour vertices is a matter of retrieving an adjacency list. Iterating through a list has complexity $\mathcal{O}(V)$, which is acceptable. Moreover, in terms of memory, for sparse graphs not too much space is occupied. Evidently, fully-connected graphs are more heavy, however, there is no need to pre-allocate memory for vertices and edges that may not even come to be inserted.

In what regards the information of the pheromone levels of each edge of the graph, which are required to be saved and monitored, needed for the *Ant Colony Optimisation* algorithm, an adjacency matrix was employed. As these value influence the path that a ant takes across the graph, these need to be updated when an ant finishes one cycle and decremented for the evaporation of pheromones along time, it is an advantage of having complexity $\mathcal{O}(1)$ for reading and writing values. Using a $V \times V$ matrix of type *double* may occupy much memory space when there is a large number of nodes. The space of a triangular matrix is enough to save all of the information, as the graph is undirected. Note, however, that this memory optimisation would not be that significant since there are usually more ants than nodes in a graph and each ant keeps an array of visited nodes. The extra space of the square matrix for the pheromones is rather irrelevant when compared to the amount of memory an ant takes.

3.2 Ant

An Ant in the very minimum should be able to save the path that has traversed so far and since is known that will be at maximum equal to number of nodes it was used a simple int array with $n+1$ positions to save this path (since it goes to the nest node twice), this would save the path by saving on the position 0 the nest and on every subsequent position the order in which they were accessed during the ant's search for a cycle. All positions with exception of the 0 are initialised at -1 and so it is easy to verify if a node was visited simply by checking its position in the array, making it $\mathcal{O}(1)$ time complexity. Furthermore, inserting another node was made simple by saving a node counter and so it consists on writing on the next node id position of the array the order of which he was accessed, making it again $\mathcal{O}(1)$ time complexity. On the hand to ensure no bugs deletion has to verify every position to check if there is a higher value than the new node counter and have it reset, being because of that $\mathcal{O}(N)$ complexity, since there are far more accesses to check if a node has been visited and insertions of new nodes to the path compared to deletions (that even if implemented as a list would require to transverse the path till the node is found, being potentially $\mathcal{O}(N)$) this solution scales well with size in terms of time and uses almost minimum memory to save the path.

With scalability in mind and to have the possibility of having different types of Ants traversing different *mazes* at the same time every ant saves its own constants and a reference to the maze that in which is trying to get a solution to the TSP problem and since this values should not change after ant initialisation they were made final.

In order to ensure randomness in the ants movement the random generator used is saved as static so that it does not happen multiple ants being initialised with the same seed and taking the same decisions.

3.3 Pending Event Container

The PEC is implemented as a simple *PriorityQueue* of Events from *java.util* setting the priority based on the time of the event to be simulated it is initialised with a observation event at time $\tau/20$ and with a move event for every ant. Every event returns the events that they origin and they are added to the PEC if their time is lower than the final time simulation τ.

4 Extensibility of the proposed solution

From the very start of the software development, even before beginning with programming, one of the objectives was to ensure that the methods were flexible and extensible. In particular, three interfaces where created for the graph. One concerning higher-level operations over the graph and two others concerning vertices and edges. These interfaces provide many methods that are, in fact, implemented and allow other developers of building their own solutions and using the most desirable data structure for their problem. Additionally, the *Graph* interface was implemented by a class containing methods for undirected graphs. This class was eventually extended with other methods necessary for the TSP problem.

Events are based on the abstract class *Event* and polymorphism is used to add them to PEC and as so it is easy to create a new type of event and to add it to the PEC directly or through other Events. With the current implementation there are 3 types of events, one for the movement of the ants, other for evaporation of pheromones by the edges of the graph and at last a event that generates observations at a specified time of simulation.

Classes are also prepared to to different simulations on the same program execution if needed having non-static methods and variables that can save multiple simulations and also have, e.g., multiple types of ants traversing the same graph on the same simulation.

5 Application performance

Since the *Travelling Salesman Problem* is a *non-deterministic polynomial time problem* (*NP*), it is not possible to have a sure best solution without testing all solution possibilities, since this proves to be impractical for even smaller number of nodes (for $n = 30$, in a fully connected graph the are $30! \approx 2.6 \times 10^{32}$ path possibilities). Additionally, as for sparser graphs, the existing edges may not be even sufficient for creating a *Hamiltonian Cycles*.

In spite of finding a combination of the various simulation that consistently provided low cost cycles, the expression that determines the probability of the ant moving from node i to node j_k,

$$P_{ij_k} = \frac{c_{ij_k}}{c_i}, \tag{1}$$

where c_i is a normalising constant, and c_{ij_k},

$$c_{ij_k} = \frac{\alpha + f_{ij_k}}{\beta + a_{ij_k}} \tag{2}$$

corresponds to cost of moving from one node to the other, which depends on the value of the pheromones f_{ij_k} in that edge and the actual cost of the same edge in the graph a_{ij_k}. This equation reveals that have a small value of β, increases the impact that the weight of the edge does on c_{ij_k}. On the other hand, a low of α, gives the pheromone level of the edge to have greater importance.

Moreover, it was concluded having a large number of ants trying to find paths following a *greedy* approach with the weights (having $\beta \ll$ weight values) to populate with pheromones the edges with lower cost that can create *Hamiltonian Cycles*. In this manner, the ants will simply try different paths almost randomly (almost disregarding weights since they are way lower when compared to the pheromones levels) and hoping that the ants found the best path or at least one with cost close to the best one. The pheromones, in sense, guide the ants to move along the *best* set of edges. From this point on, the ants transverse the edges marked with pheromones, attempting various combinations of edges which result in trying to find the cheapest *Hamiltonian Cycle*. Notice that increasing the number of ants corresponds to a linear growth of the run-time complexity of the simulation, and, therefore, when compared to adding one more node to the path which results in a factorial complexity growth of the simulation time, is fine.

In this fashion, the problem is divided in two parts:

1. Following a greedy strategy to tries to find all possible paths with the lower costs edges which can be adjusted with the β value.
2. From the edges with lower costs that form paths, the ants try to find random paths with the objective of achieving a better solution. The beginning of this second phase is adjusted by varying the values that change the pheromones behaviour, whether by how many pheromones are dropped on the edge cycles, by varying γ, or by changing the time in between evaporations, η, by changing how many pheromone are, in fact, decremented at every evaporation, ρ, finally, the choice with α also controls the influnce of pheromones.

In both parts the greatest factor of success is the number the number of ants since they they are almost selecting random paths. Thus, having a larger colony of ants increases the number of paths tested and specially for the first part it is the uttermost importance to find most or at least a really good portion of the possible paths, since edges not found will almost certainly not be checked in the second phase, for which the pheromones govern the movements of each ant.

One obvious problem of this approach is if some path has some edge that has a high cost but a better overall cost it may be skipped in the first phase and making it impossible for the second to find the best path. Even knowing this problem, the test results showed that this approach yielded very good results when compared to attempts to fine tune the hyper-parameters of the problem, which revealed to be very troublesome and impossible without the use of run-time debugging to check pheromones levels of the the edges after observations and even then the simulation would take more time τ to reach moderate good results with no consistency, especially when tested against random graphs with the same number of nodes and same level of connectivity (but different weights).

In order to assess the performance of the proposed solution, five concrete tests were conducted. The main objective of these tests was mainly to analyse how the results of the simulation varied with change of graphs connectivity (all with 30 nodes) when following the approached strategy, and as such the following tests were created:

- test_1: Fully connected with totally random weights from 1 to 30
- test_2: Fully connected with a 30 path cost created by us (path goes from 1...30) random weights from 1 to 10
- test_3: $\approx 50\%$ connected with random weights from 1 to 30
- test_4: $\approx 25\%$ connected with random weights from 1 to 30
- test_5: $\approx 75\%$ connected with random weights from 1 to 40

In all of the tests the approach explained before provided better best paths estimates when compared to fine tuning the hyper parameters by hand and not having a second phase that would ignore the weights of the edges. It should be noted that all tests use the same hyper parameters.

6 Conclusions

It was created a Java application that tries to get a solution to the *Travelling salesman problem* following object oriented programming ideals of expandability of the code and easiness to implement new features on top of existing ones. To achieve this goal there were used interfaces to create bases that would support the rest of the code, abstract classes in order to implement polymorphisms so that, e.g., the PEC didn't had the necessity of knowing what type of event would receive being to accept every type.

It was created documentation so that searching for classes and methods and its functions could be eased by explaining with moderate detail the behaviour of every method and how they should be used.

The data structures that were used to save the data were thought before even starting programming (with the UML) so that the time complexity of the simulation could be minimal without having to much memory wasted. Using references to objects inside others whenever possible made so that there was no memory duplication of the bigger structures, leaving constants as non-static in order to have the possibility of expand the code to support multiple simulations being run at once.

In terms of the simulation, for different hyper-parameters it was found that it was better to use a two phase approach with many ants ($\gg$ number of nodes) where first the ants would do a *greedy* search of edges with low costs that would create *Hamiltonian Cycles* and then populating those edges with high levels of pheromones so that the weights would be neglected and the ants would simply try to randomly find cycles in hope of finding one close the best one. This approach proved to be reliable and consistent when comparing to hand tuning of the hyper parameters of the simulation, finding better paths with less simulation time (real and τ).