INSTITUTO SUPERIOR TÉCNICO

OBJECT ORIENTED PROGRAMMING

TRAVELLING SALESMAN PROBLEM BY ANT COLONY OPTIMIZATION

Project Report

May 9, 2019

GROUP 7

ION CIOBOTARI, 84075

PEDRO FERNADNES, 84168

IVAN ANDRUSHKA, 86291

Contents

Chapter 1

Introduction, Objectives and Methodology

1.1 Introduction

The objective of this project is to implement the Ant Colony Optimization algorithm to solve the TSP problem. That is, to find Hamiltonian cycles inside a graph. This algorithm was implemented following the regular guidelines defined for this problem. The only peculiar choice was to build a graph based on nodes, but each node contains a list of edges that are defined by it's starting and finishing nodes. This changes close to nothing, except for the methods that require access to nodes, which now use edges to get the nodes.

1.2 Data structures

The choice of data structures was made with regards to complexity and to the type of operations that need to be made on the data.

TspEdge: This is the basic element used throughout the project. It contains an Array of integers of size 2 that keeps the values of the nodes that are connected by this edge.

TspNode: This class contains an ArrayList of Edges to which this node is connected. The reason this was chosen over a LinkedList is because we need to access the data frequently, which is an O(1) operation in this type of structure rather than the O(n) complexity of accessing an element in a LinkedList. The argument for why an ArrayList over a regular Array is that the edges are inserted in the structure as we read the input file, making the insertion more efficient as we cannot predict how many edges each node will contain.

TspGraph: The graph is an ArrayList of nodes, making it a dynamic matrix. The reasoning behind this choice is the same as explained above for the Node class.

Ant: This class contains its own ArrayList that stores all the edges that the ant has traversed so far. This allows the ant to keep track of the edges that have been visited and compute the respective probability to visit new edges. If the ant computes a cycle that does not correspond to a hamiltonian cycle, this cycle is removed from the ant path.
PEC: The pending event container is a LinkedList. This was chosen because we do not need to access intermediate values inside the PEC, only the first, meaning the access complexity is always O(1). The problem with the PEC is that the insertion is sorted, so the complexity is O(n) even though the LinkedList is still the most efficient in this regard. Thankfully, the iterator class exists which was used to keep a pointer to the correct position, making the

insertion actually O(1).

1.3 OO solution

The project was divided into 5 packages: pcolony, pevents, pgraph, pmain and psimulator.

pgraph: contains the basic classes required to implement the graph and the methods that act on its structure: TspEdge, TspNode and TspGraph.

pcolony: contains the Ant class that generates the number of Ant objects that form the colony.

pmain: contains the Main method and the UserHandler method which overrides the regular ContentHandler used to parse the xml files, which was done using SAX.

psimulator: package contains the PEC Interface that defines the Simulator class. This class is responsible for executing the events present in the simulator list.

pevents: contains all the event classes, those being AntMove, EvaporatePheromone and Observation. The only polymorphism used in the code is for the Event class which is disjointed into the 3 types of events that exist.

1.4 Performance

The theoretical complexity analysis for this kind of probabilistic algorithm is an untreatable problem. So, rather than analysing time complexity, we analysed the convergence based on the parameters used. The alpha and beta parameters alter the probability of traversing an edge Pij. δ changes the time it takes to traverse an edge. By increasing δ, the time increases, leading to less events overall. On the other hand, by lowering the value of δ there are more events. This means that we need to find an appropriate value for δ given the overall simulation time. For example, if we run a simulation that lasts 1 second with a δ of 1, there will never be any paths found, since there are not enough events. On the other hand, if we change δ to 0.0001 there are more than enough events to find a path. Obviously this depends on the graph used. We used the graph given in project example to verify these properties. η alters the time it takes for an evaporation event to occur. This leads to less evaporations during a simulation. For the example given in the project, by using η=2 there are, on average, over 5 simulations, 931 evaporation events. By changing η to 10, the number of evaporation events changes to 208, confirming the conjectures made about this value. ρ changes the amount of pheromones that taken from the edges when an evaporation event occurs. Again, using the example with ρ=10 we get 931 evaporation events, while with ρ=100 there are only 671. This makes sense, since the pheromones are quickly evaporated, leading to fik=0. When this

happens we need to wait until an ant finds a new path in order to lay new pheromones. This leads to less evaporation events overall. In contrast, γ changes the amount of pheromones that are laid down by the ant when a Hamiltonian cycle is found. Using γ=0.5 there are again 931 evaporation events, while with γ=10 there are 1025 events. Once again, this is a reasonable result since we now require more events in order to evaporate the pheromones that have been laid down, leading to less downtime (downtime meaning fik=0, since no evaporation events are created in this case) between new events.

Chapter 2

Test Analysis

All of the following tests were made using the parameters: α=1, β=1, δ=0.2, η=2, ρ=10 and γ=0.5.

2.1 Test1

The first test done used a graph that contains a node that connects to only one of the other nodes (fig. 2.1), using 1000 ants and 30 seconds of simulation time. This means that in order to complete a Hamiltonian cycle, the ants would have to go this isolated node and then back, which is not possible due to the fact that an ant can only visit the same node once. In this situation, the algorithm correctly returns no cycles as was expected.

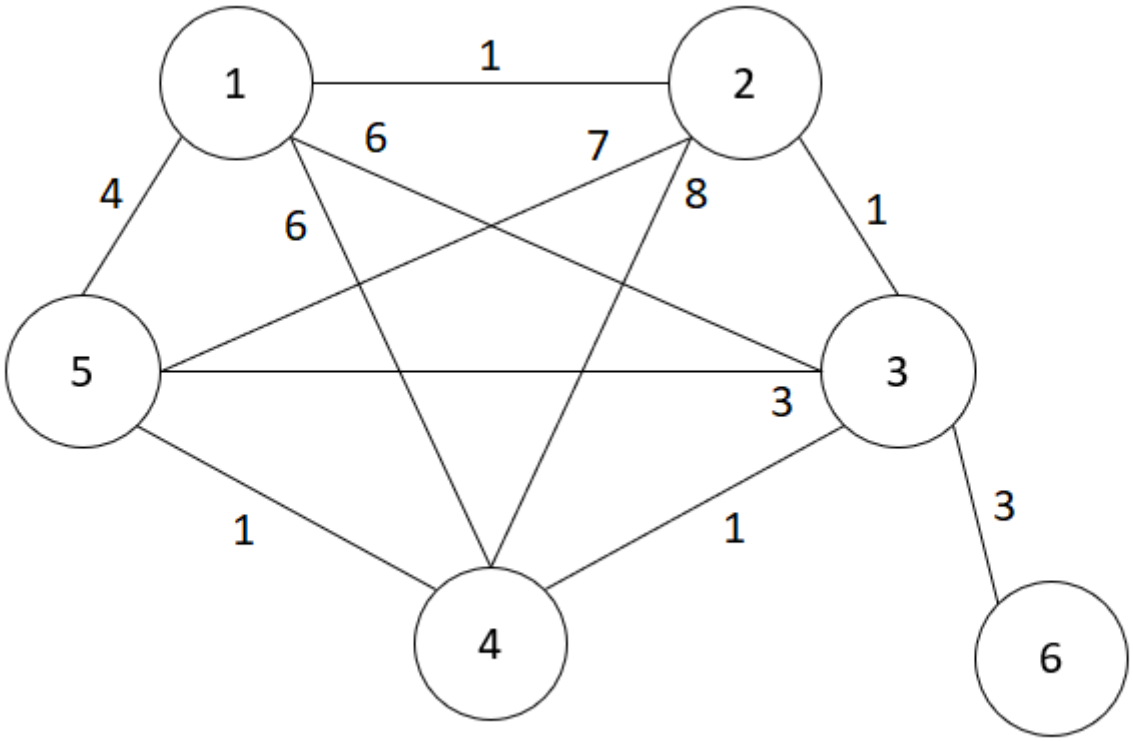

Figure 2.1: Graph with isolated node

2.2 Test2

This test is simply a large and very dense graph, with 50 nodes and all nodes connected between themselves with random weights. Since the graph is so large and dense, it requires a certain amount of time and ants in order to find a path. Running the simulation for 50 seconds

with 10000 ants, a path was only found after 37.5 seconds, which is the observation 15/20. This does take a while in real time, but it is reasonable considering the amount of ants and the density of this graph.

2.3 Test3

The third test is a circular graph (fig 2.2). This was simulated during 300 seconds with 200 ants. There are 2 possible solutions for this problem. The clockwise and anticlockwise paths. When subject to this test, the program returns one of these solutions, depending on which it finds first, which is what we expected.

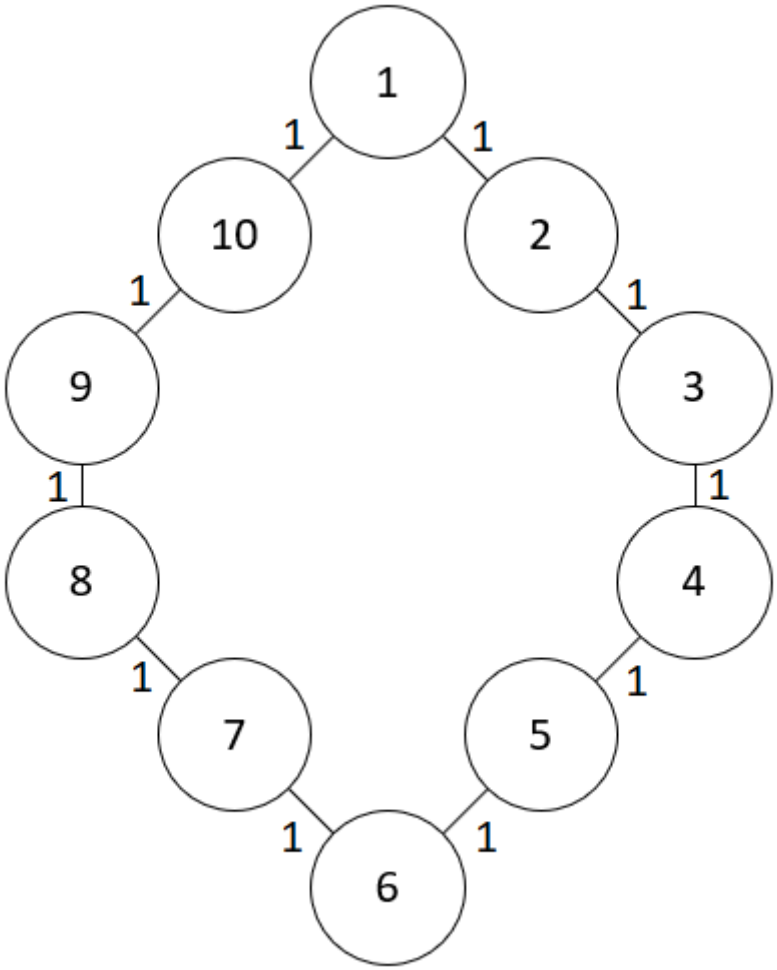

Figure 2.2: Circular graph

2.4 Test4

This test was conducted using a graph with all nodes connected between themselves with all the edges having weight 1. This means that all possible paths have the same total weight, so every simulation will return different results. This is indeed what was observed.

2.5 Test5

This final test involved a graph with all nodes interconnected, where the outer circle contains edges with low weight, while the inside was filled with heavy edges. This means that, while the ants do have many options, the outer path is by far the best and most likely course. With 200 ants and 300 seconds of simulation time, since the probability of this outer path is so much higher, the first observation already returns the best path. On the other hand, if we simulate this graph with just 1 ant we can see different paths being chosen at first, but it almost always ends up finding the best path.