Travelling Salesmen Problem by Ant Colony Optimization

João Lopes, Gonçalo Carvalho and Alessio Vacca

May 2019

1 Introduction

For this project we were asked to implement in Java an Ant Colony Optimization algorithm to solve a Travelling Salesman Problem. The task was model the problem using UML and to implement the model in a Java application. Our main focus during the project development was to provide a Java solution implementing an extensible and reusable framework.

2 Data Structures

2.1 Graph

Since our problem aims in finding the shortest path in a weighted graph we needed to create a data structure to implement a weighted graph. In order to have a good separation between our implementation and a generic weighted graph we created an interface for a generic weighted graph. This interface gives necessary methods for an object of a class that implements this interface to communicate with other classes.After creating the interface for a generic graph that has nodes and edges we implemented the class Graph.Java itself and a class Edge.Java.

For the data structure to implement the graph we had two possibilities, an adjacency matrix or an adjacency list. To decide which structure to use we have analyzed the complexities of both structures in the different cases. An adjacency matrix has as main issue the memory it takes to store the graph. For any graph the memory that it takes to store the graph is always N^2 (being N the number of nodes). For graphs with a large number of nodes this is a big concern. On the other hand an implementation with adjacency matrix can be faster when realizing operations with the edges of the graphs. For example to check if two nodes are connected the complexity is always 1. The main advantage with the adjacency list implementation is that the memory it takes to store a graph can be much lower. For sparse graphs the memory that the structure uses is $N + E$ (being N the number of nodes and E the number of edges), since in a sparse graph we have $E << N$ the memory used is almost the square root of the memory used by the adjacency matrix. The disadvantage of the adjacency list lays on doing operations with the edges, for example to check if two edges are connected in the worst case the time complexity can be N.

Given that the problem we want to solve aims on finding a cycle that goes thought all the nodes (only once) we expect that the most interesting application of this problem is to find those cycles in large graphs with few edges. For those graphs the most adequate data structure would be adjacency list so that was the structure we used to implement our graph.

Our graph is then an ArrayList of LinkedLists as shown in figure 1.

2.2 PEC

The core of the simulator is the Pending Event Container (PEC). To implement the PEC we first created an interface PendingEventContainer that provides the necessary abstract methods for any PEC implementation to work. Those method are responsible to add new events to the PEC, to remove events, to check when the PEC is empty and finally to give next event to be simulated. We then created a class PEC.Java to implement the interface. Since the main reason to have a

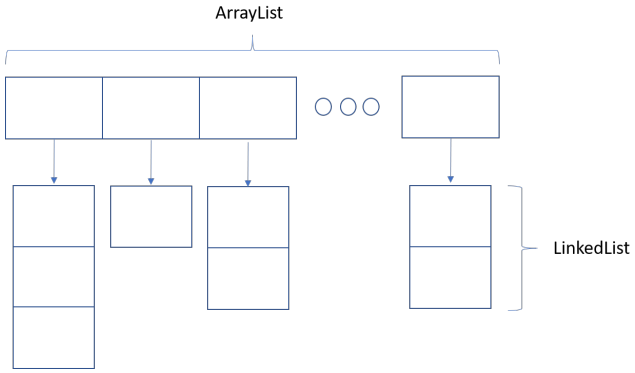

Figure 1: Graph as a Adjacency List

PEC is to have the Events to be simulated sorted we had to use another data structure to have the Events stored by their time stamp. The data structure that we used was a PriorityQueue from java.util. Since we are sorting non primitive elements we must provide the priority queue or comparable types (types that implements the comparable interface) or a Comparator when creating the priority queue. Given that the only elements in the simulator to be sorted are the Events we decided to create a class Comparador.Java that implements the Comparator interface from java.util to compare DiscreteEvents. DiscreteEvents is also an interface that are implemented by the three different Events in the simulator this way the compartor can be used to sort objects from different classes.

3 Interfaces and Abstraction

In order to make our code extensible we implemented 6 different interfaces and one abstract class.

- The first interface we have implanted was XmlData. This interface basically provides all the necessary abstract method to set and get the parameters that are indispensable for the simulator to work. Those parameters can be in a file of any type as long as the person who is going to implement the interface provides all the methods in the interface.
- The interface WeightedGraph provides all the necessary abstract methods to initialize and realize the basic operations with a weighted graph. If someone is implementing a different weighted graph to be used in this application it's free to choose the desired implementation (ex: adjacency list, adjacency matrix etc..) the only thing to be assured is that all the methods defined in the WeightedGraph interface are implemented .
- The DiscretEvent interface is the interface to be implemented by the different Events in our application. The only abstract methods in the DiscretEvent are the methods simulate() and getTimeStamp(). For that reason, anyone who is going to reuse our code is free to create new events as long as the new events implements this interface the application will continue to work.
- The PendingEventContainer interface has the basic abstract methods for a class that is implementing this interface to communicate with the other classes of the application. A class that wants to implement a PEC needs to implement the abstract methods in this interface for the application to work. Those methods are addEvPEC() to add new elements to the PEC, removeEvPEC() to remove events, nextEvPEC() to provide the next event in the PEC, isEmpty() to check if the PEC is empty.
- The Traverser interface is the interface to be implemented by the class that provides the objects that will traverse the graph. This interface has only three methods move() to move

the traverser, chooseNextNode() to choose the next node and finally getPath() to get the path already made by the traverser. In our application the class Ant.Java is implementing this interface but a different class may implement this interface.

- The TraverserPath interface in our application is implemented by the class Path.Java. This interface provides the method for a class that is going to store the path done by an object from a class that implements the interface Traverser. The methods are both the setters and getters for the visited nodes and correspondent weights.

4 Colony Simulator

The ColonySimulator.Java is the class where the main method is implemented. Is in the Class ColonySimulator that the data from the xml is initialized, so is the graph, the PEC and the list containing all the ants in the colony. The simulator itself is a while cycle where in each iteration the next Event in the PEC is simulated. Every time that the event simulated is an Event that moves an Ant (EvAnt_Move) we check if the correspondent Ant has already found a Hamilton Cycle, and if so checks if it is shorter than the shortest one already found. The shortest Hamilton Cycle is then stored in an object of class Report.Java. It is also inside this loop that we keep counting the number of simulated events and again storing those variables inside an object from the class Report.Java.

5 Results

5.1 Test 1

In this test we created a fully connected graph with 100 nodes with all the edges with weight 1. In this graph all the paths are optimal and since this is a stochastic simulation the path found was, as expected, different every time we ran the simulation. We also did this simulation to test the performance of out simulator in the worst case possible. Our graph is implemented as an adjacency list so a fully connected graph will lead to the worst performance of our application. As explained before, since we are trying to solve a TSP we do not expect our application to deal with very dense graphs, so this is not a big concern. Nevertheless the application took only 7.45 seconds to run the simulation with the following parameters:

- finalinstat = 200, antcolonysize = 200, plevel = 0.5
- nbnodes = 100 nestnode = 1
- alpha = 1.0 beta = 1.0 delta = 0.1 eta = 2.0 rho = 10.0

And realized 200479 ant move events and 76775 evaporationn events.

5.2 Test 2

In the second test we created a random graph with 100 nodes and each node with 10 to 20 connections to random nodes. All the connections in this graph were seted with weight 10. After this we set connections between successive nodes with weight 1. So that the shortest Hamilton cycle in this graph is the cycle {1,2,3,4....99,100,1}. We run the simulation with the following parameters:

- finalinstat = 1000, antcolonysize = 10000, plevel = 0.2
- nbnodes = 100 nestnode = 1
- alpha = 1.0 beta = 1.0 delta = 0.1 eta = 1.0 rho = 2.0

After running this simulation the optimal Hamiltonian cycle was not found but the one found has a weight that is very close to the minimum possible.
The probability of picking the next node is inversely proportional to the weight of the edge associated but this probabilities as shown in section 2 of the project description are normalized. So even though for each node one of the adjacent edges has weight ten times lower than all the others the probability to pick this node is not 10 times larger than picking one of the "wrong" (with higher cost) nodes. In fact if the number of adjacent nodes is to large the probability of not to choose the shortest edge can be lower than picking the shortest edge. Also for a graph like this one with, 100 nodes, it is very unlikely that for every node the decision taken is the one associated with the lower edge weight. As shown in the results even with 10000 Ants the shortest cycle was not found. Nevertheless for a big graph like this the ACO algorithm can be useful to find a solution very close to the optimal one.

5.3 Test 3

The third test is a graph that has more than one hamiltonian cycles (bidirectional). The lowest weight path has a difference from the next lower of 1 weight. During our simulation when using the default values it does not find the best one but the 2nd best. However when we decrease the delta parameter, the interval between moves is reduced, so it finds the best hamiltonian cycle most of the times.

5.4 Test4

The test4.xml graph is a graph that has 81 nodes and only one hamilton cycle (bidirectional). We tried different approaches to find this cycle, starting with the time of simulation for default values and changing the time doesn't change anything and it doesn't find the hamiltonian cycle. There's a relation between the three parameters, the time of simulation allow the program to have more evap/move events. Changing the value delta will reduce the time between events so more events will happen during the finalinst time. As it never finds an hamiltonian cycle when changing the the finalinst. We changed the value of delta for different values for a fixed antcolonysize of 2000 ants:

delta	mEvents	evapEvents	Hamiltonian cycle
0.2	489066	-	-
0.02	4883427	-	-
0.002	48606214	-	-

This graph has 3 big triangles(with small ones inside), only connected to each other by only one node, and this makes the hamiltonian cycle really hard to be found, because if it gets to the connection from a wrong node it will never find the Hamiltonian cycle since the chosen of the new edge is ruled by probabilities. In a last effort for our program to found the hamiltonian cycle we increase the number of ants for 20000 keeping delta with value 0.002. This configuration took a lot of time to run and it did not found the hamiltonian cycle.

5.5 Test 5

This graph has 16 nodes connected in a triangular way. All the edges have the same weight except in one of them. For a low value of finalinst = 20 there are a lot of times that the program does not finds the hamiltonian cycle. For that, increasing the value of the simulation will be enough to find it. As this is a stocastic method and the graph is relatively small and with few edges between them, with enough finalinst it will always find the best hamiltonian cycle.

6 Conclusion

Using a ACO to solve a TSP can be a good approach in many cases but the user might be alert that depending on the graph it might be unlikely to find the optimal solution without a big computational effort and without very specific simulator parameters. Though if we are dealing with graphs with multiple possible Hamiltonian cycles this method can be very useful to give a solution that even if it is not the optimal it is still very close to it.
This project was implemented to be the most extensible possible, making use of the open-closed principle. Every communication between different entities is done through interfaces so that if it is needed to change implementations the application still works properly, for example changing graph implementation from adjacency list to adjacency matrix.