Intelligent systems lab assignment

Project documentation.

Index

Setting up the project.

The following documentation will be used to serve as guideline and to show important aspects related to the implementation of this project.
The project consists on the implementation of an agent program capable of finding the optimal route for a vehicle that circulates through a set of places of a town.

The town information will be given by a graphml file, the project will be implemented in java using eclipse and will consist on a set of steps or tasks that progressively will lead to a fully functional program.

This documentation can be used to modify aspects of the project or as guideline for new implementations.

Graph representation.

In order to allow the program to operate optimally it is necessary to first load the graph (given as an input graphml file) into memory, for this purpose we will use appropriate data structures like ArrayList or HashTable.
In an early state of the development we will use ArrayList because of the simplicity it gives to the implementation, the use of different data structures will be considered in a future approach when more aspects of the nodes and edges will be necessary to take into account.

For the task 2 the use of Hashtable was considered instead of using ArrayList for the edges and nodes storage, the use of this data structures allows us to search and pick node and edge elements in a faster and smarter way due to the use of “key” values.

```
private Hashtable<String, control.Node> graphNodes = new Hashtable<String, control.Node>();
private Hashtable<String, Edge> graphEdges = new Hashtable<String, Edge>();
```

The image above shows how both edges and nodes are being stored after the task 2 implementation, changes in graph methods due to the use of Hashtables will be shown later in this document.

State space and states representation.

In the task 2 the need of implementing an state space representation was introduced, this need also carries the need of implementing an state class in order to represent states that form the complete state space.

The state space contains methods for the successors processing (for a given state) and a control method to check if a given state is possible inside our state space, the constructor of this class takes the graphml file as input and it creates and contains the representation of the graph and also a Hashtable that contains all the nodes of the graph as a set of possible states, this class is needed for its usage in the problem domain.

The State class represents an state and it uses just the id of the nodes in order to store the information instead of using the whole control.Node object that was implemented in Task 1.

```
public class State {
    private String node;
    private PriorityQueue<String> listNodes;
    private String id;
```

Image above shows the structure of the State class, it's so simple but its so useful it's important to mention that the initial state is given to the program through a JSON file, the JSON parser was also implemented in task 2 and will be shown in this document later, the id value of the state represents an MD5 string.

Frontier representation.

Frontier class represents as its name says the frontier of the search problem, it is implemented through a PriorityQueue and its implementation just replicates the methods of this data structure.

The important aspect of this class is the usage of the TreeNode class that will be discussed later in this paragraph.

Also its important to mention that the data structure to be used to implement the frontier was chosen by the usage of an stress test which results will be shown in the results per Step/Task paragraph of this document.

TreeNode.class.

In order to represent elements on the search graph we decided to use the TreeNode class, this class contains a control.Node object in order to represent the parent of the current node.

Also it contains the current state represented as an State object, the actual cost of the path, the cost of the node, the depth of the node and the action that was taken on this node, it also implements a comparable in order to sort TreeNode elements that are contained in the PriorityQueue that represents the frontier.

```
public class TreeNode implements Comparable<TreeNode> {
    private control.Node parent;
    private State currentState;
    private double cost;
    private String action;
    private int depth;
    private double f;
```

Note that for the initial state there is no possible parent node so the value of that field will be null in the case of the initial state.

```
@Override
public int compareTo(TreeNode o) {
    if (f < o.getF()) {
        return -1;
    } else if ( f > o.getF()) {
        return 1;
    } else {
        return 0;
    }
}
```

The image above shows the implementation of the comparable for the TreeNode class in order to sort elements in the frontier.

Problem representation.

The problem is given initially as a JSON file, it contains the name of the graphml file to be readed and a State object that represents the initial state.

The only important aspect to mention of the problem is the isGoal method that will be shown later in this document.

Reading XML files.

First of all it is necessary to read XML files, for this purpose we will import the package javax.xml.parsers.DocumentBuilder as DocumentBuilder is used for the purpose of reading XML documents.
There is also a problem with this kind of graphml files, the keys that specify each attribute of the nodes (or edges) are variable, this means that the documents are different from each other and they use different key variables to assign data.
To solve that problem the following lines of code are used:

```
NodeList kList = doc.getElementsByTagName("key");

for (int i = 0; i<kList.getLength(); i++) {
    Node kNode = kList.item(i);
    Element kElement = (Element) kNode;
    if(kElement.getAttribute("attr.name").equals("name") && kElement.getAttribute("for").equals("edge"))
        NAME_KEY = kElement.getAttribute("id");
    else if(kElement.getAttribute("attr.name").equals("length") && kElement.getAttribute("for").equals("edge"))
        LEN_KEY = kElement.getAttribute("id");
    else if(kElement.getAttribute("attr.name").equals("x") && kElement.getAttribute("for").equals("node"))
        X_KEY = kElement.getAttribute("id");
    else if(kElement.getAttribute("attr.name").equals("y") && kElement.getAttribute("for").equals("node"))
        Y_KEY = kElement.getAttribute("id");
    else if(kElement.getAttribute("attr.name").equals("osmid") && kElement.getAttribute("for").equals("node"))
        OSMID_KEY = kElement.getAttribute("id");
}
```

As shown in the image, the kList is a list of type NodeList (part of the package org.w3c.dom) that contains all nodes that use the tag “key” in the graphml document, using this code we are iterating over all the nodes with the “key” tag commonly located in the head of the document that contains information about the assignation of “key” variables to the nodes and edges information.
The process takes each element and checks if it is related with a determined variable, for example as shown in the image in order to assign the key value for the OSMID of a node we

check if the selected item refers to the osmid value in the field “attr.name” and also if it is part of a node, then the variable OSMID_KEY is assigned to the value of the id that will represent the OSMID in the XML document.

Finally after all the key elements for reading a node correctly are assigned the reading process of a node is shown in the image below:

```
for(int i = 0; i<nList.getLength(); i++) {
    Node nNode = nList.item(i);
    if(nNode.getNodeType() == Node.ELEMENT_NODE) {
        Element eElement = (Element) nNode;

        data = eElement.getElementsByTagName("data");

        for(int j = 0; j<data.getLength(); j++) {
            Node dNode = data.item(j);
            Element dElement = (Element) dNode;
            if(dElement.getAttribute("key").equals(X_KEY)) X = Float.valueOf(dElement.getTextContent());
            else if(dElement.getAttribute("key").equals(Y_KEY)) Y = Float.valueOf(dElement.getTextContent());
            else if(dElement.getAttribute("key").equals(OSMID_KEY)) ID = dElement.getTextContent();
        }
        control.Node n = new control.Node(ID, X, Y);
        //System.out.println(n.ToString());
        graphNodes.add(n);
    }
}
```

In the image appears how a node is readed, there is no need of showing how the edges are readed because the process for both nodes and edges is similar.
As shown in the image, we iterate over a nList which is a NodeList that contains every node [1]of the graphml file, then another loop iterates over the data elements contained in that node, this second loop is where the “key” values assigned before are used to assign the values for the variables of the object node which will represent the node in our program, after all this process the new control.Node object is added to the list of nodes graphNodes, the process for edges is exactly the same but the list of edges is referred as graphEdges.
Notice that the object Node appears as control.Node, this is caused by the already existing Node class of the package org.w3c.dom in order to differentiate between them.

In the task 2 a little change was introduced to the XML reader that is now it also reads the OSMID of edge elements in order to retrieve edges from the Hashtables introduced in the Graph class after task 2, the functionality of the XML reader is the same than in previous tasks.

Reading JSON files.

In order to obtain the initial state of the problem and the graphml file to be readed we need to parse a JSON file, the parsing of JSON files only needs the objects that are represented in that file to be implemented exactly as these are shown in the JSON file.

[1] In this context, a node is referring to a graph node that contains an ID and the coordinates of that node in the OSM map, this can be confusing due to the use of NodeList where the word node refers to a node of the XML document that contains data about a determined element.

An example JSON file that we will use in this program could be like the one shown in the picture below.

```
{
    "graphlmfile": "Ciudad Real/data/Anchuras.graphml.xml",
    "IntSt": {
                "node": "4331489739",
                "listNodes": ["4331489528", "4331489668", "4331489711", "4762868815",
                "4928063625"],
                "id": "f4b616551965fb586e608397c308bf0f"
             }
}
```

The implementation of the JSON parser uses the package com.google.gson.Gson in order to parse JSON files like the one shown above, the parser is implemented in our code in a very simple way.

```
package control;

import java.io.BufferedReader;

public class ReadJSON {

    public static Problem Read(String jsonFile) {
        Problem init = new Problem();
        Gson gson = new Gson();
        try {
            BufferedReader br = new BufferedReader(new FileReader(jsonFile));
            init = gson.fromJson(br, Problem.class);
        }catch(Exception e) {
            e.printStackTrace();
        }

        return init;
    }
}
```

As can be seen in the image we just call the class imported in the com.google.gson.Gson[2] using an object called gson and using BufferedReader with the JSON file as argument, the JSON parser was implemented in an independent class for the sake of simplicity.

[2] The version used for the gson package in our program is gson-2.8.5 there was a problem with the java version in terms of dependencies for packages that was solved modifying the module-info.java file of the project.

Methods.

In this section all methods used in the program will be explained briefly.
Auxiliar methods and other stuff that is not specified directly in the project specification may not be shown.

Graph.BelongNode(String id).

```
public boolean BelongNode(String ID) {
    for (int i = 0; i<graphNodes.size(); i++) {
        if(graphNodes.get(i).getID().equals(ID)) return true;
    }
    return false;
}
```

Simple and clean method use to check if a node is already in the graph we are working with, this method will appear in some other methods as a preventive measure to prevent errors related with unknown nodes in the graph.
The implementation iterates over the list of nodes and checks if the provided node id is in the list.

Graph.positionNode(String id).

```
public float [] positionNode(String ID) {
    float[] xy = new float[2];
    control.Node n = null;
    if(BelongNode(ID)) {
        for (int i = 0; i<graphNodes.size(); i++) {
            if(graphNodes.get(i).getID().equals(ID)) {
                n = graphNodes.get(i);
                break;
            }
        }
        xy[0] = n.getX();
        xy[1] = n.getY();
        return xy;
    }
    return null;
}
```

This method returns a 2 elements array where the first one represents the X coordinate of a node and the second one the Y coordinate.
The implementation checks first if the provided node is in our graph and then proceeds to retrieve it from the list, then each coordinate is assigned using the control.Node.getX() and control.Node.getY() methods and the filled array is returned.

Graph.adjacentNode(String id).

```
public ArrayList<Edge> adjacentNode(String ID){
    ArrayList<Edge> adjacents = new ArrayList<Edge>();
    if(BelongNode(ID)) {
        System.out.println("\nAdjacents of " + ID + ": \n");
        for(int i = 0; i<graphEdges.size(); i++) {
            Edge e = graphEdges.get(i);
            if (e.getSrc().equals(ID)) {
                System.out.println(e.ToString());
                adjacents.add(e);
            }
        }
        System.out.println();
        return adjacents;
    }
    return null;
}
```

This method returns a list of adjacent nodes regarding a provided node id, it uses an ArrayList of Edges that will be filled with the Edges where the source control.Node is equal to the control.Node provided.
The implementation checks first if the provided node exists in the graph, then it iterates over the graphEdges list and checks if the source of the Edge object is the same as the provided node, then if both are equal it adds the Edge to the adjacents list, when the process ends the list of all adjacent edges is provided.

StateSpace.Successors(State st).

Results per step/task.

In this section all results obtained per step will be shown as all the conclusions and modifications that are taken after each step.

Step/Task 1.

In this step we were ask to represent the graph and to implement methods BelongNode(String id), positionNode(String id) and adjacentNode(String id).

Results obtained in this step are the following:

```
XML file was readed in: 197 ms

BelongNode() executed in: 0 ms
positionNode() executed in: 0 ms

Adjacents of 4331489709:

(4331489709, 4331489708, Calle de la Iglesia, 46.118)
(4331489709, 4331489716, Plaza España, 17.119)
(4331489709, 4331489544, Plaza España, 25.208)
(4331489709, 4331489549, Calle Amirola, 137.537)

adjacentNode() executed in: 1 ms
```

Notice that the time spent in the execution of methods BelongNode(String id), positionNode(String id) and adjacentNode(String id) is not really relevant as it takes less than 1 ms to complete the execution and ns times won't be considered.
But the situation is different for the method that reads XML, we obtained a medium execution time of 250 ms (197 ms in the image shown) which under our point of view is an acceptable execution time for our first approach.

In conclusion, first step implementation was simple and easy to implement without execution time problems although changes on data structures will be needed in future approaches.

Step/Task 2.

In this step we were asked to implement some of the main elements concerning a search problem, the most important and representative part of this step was the selection of a good data structure to manage the frontier.

The characteristics that the chosen data structure should have are:

- Good insertion time.
- Good capacity.
- Fast management of insertion/sorting with lots of elements.

The obtained results after the tests were the following:

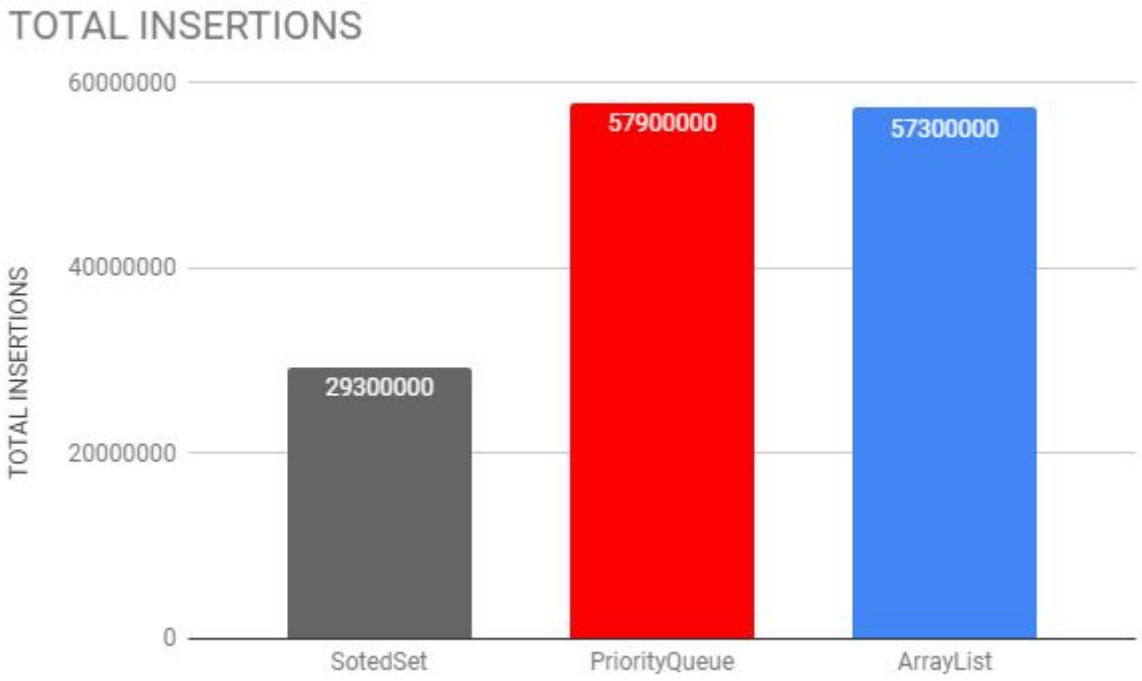
	MIN	MAX	TOTAL TIME	TOTAL INSERTIONS	MEAN
SotedSet	1	761	73038	29300000	0,002492764505
PriorityQueue	1	292	3647	57900000	0,000062987910
ArrayList	1	1557	4487	57300000	0,000078307155

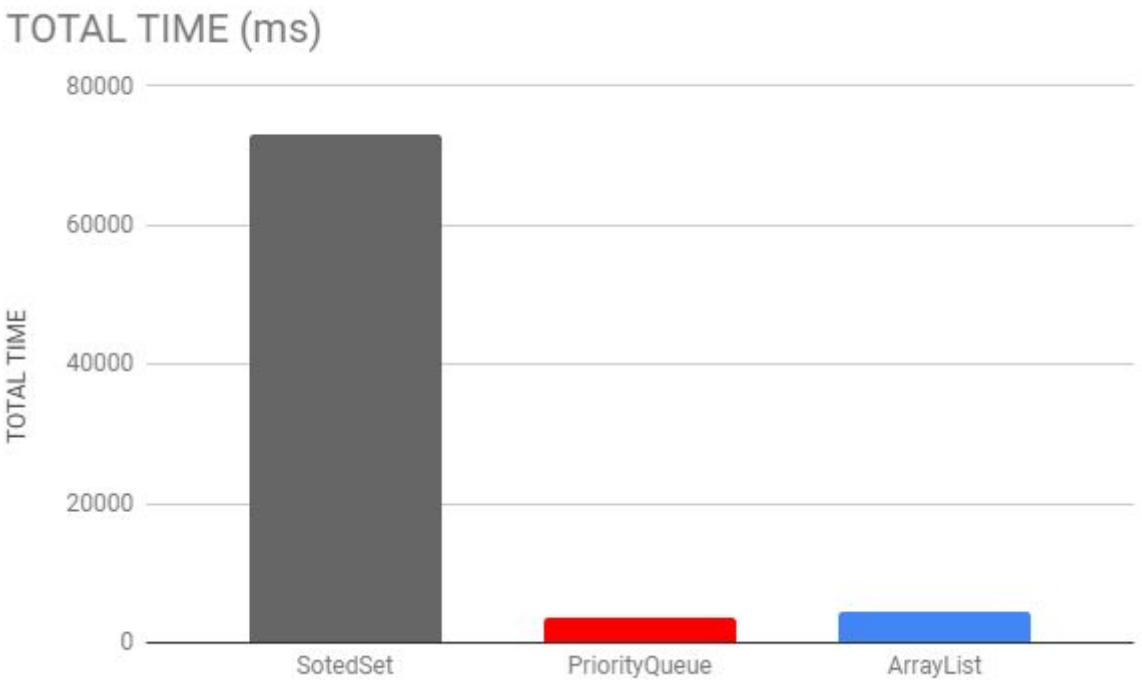
As can be seen we selected three data structures to be tested, SortedSet, PriorityQueue and ArrayList, notice that ArrayList is not a sorted data structure but with the help of the java.collections.sort method it can become a very powerful data structure in terms of sorting and handling elements (almost as good as PriorityQueue).

```
public void testList() {
    while (true) {
        try {
            TreeNode nodo = new TreeNode();
            ini = System.currentTimeMillis();
            listNode.add(nodo);
            fin = System.currentTimeMillis();
            elapsed = fin - ini;
            if (elapsed < min && elapsed > 0) {
                min = elapsed;
            } else if (elapsed > max) {
                max = elapsed;
            }
            val += elapsed;
            if (counter % 100000 == 0) {
                ini = System.currentTimeMillis();
                Collections.sort(listNode);
                fin = System.currentTimeMillis();
                printResults(min, max, counter, val);
                System.out.println("Sorting time for " + counter + " nodes : " + (fin-ini) + " ms");
            }
            counter += 1;
        }catch(Exception | Error exception) {
            printResults(min, max, counter, val);
        }
    }
}
```

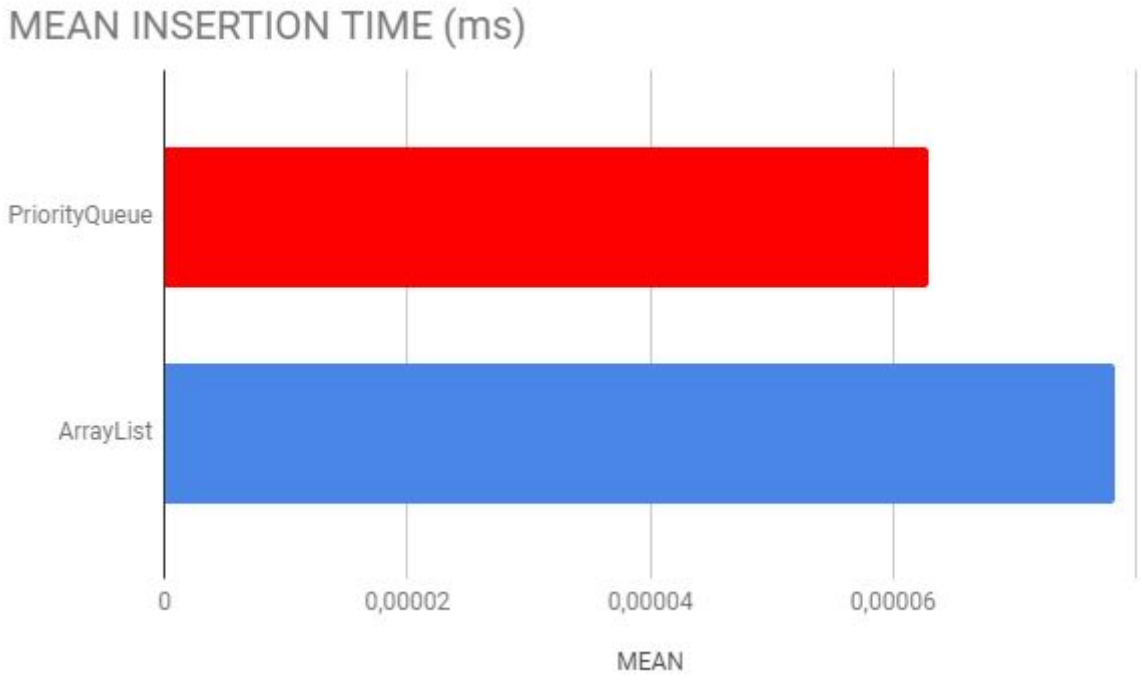
The image shows just the implementation of the test for the ArrayList due to the usage of Collections.sort method, the final time shown in the table is the sum of the insertion time plus the method Collection.sort call.

To summarize the results obtained we can generate some histograms.





As shown in the histograms we can totally discard the use of SortedSet so the final decision is based on the mean insertion time:



As we can see the selected data structure will be PriorityQueue but the performance of ArrayList is quite impressive.

Also the fact that PriorityQueue sorts when inserting without the need of an external method makes this data structure perfect for our purpose.