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.

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.

[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.

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.

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.