INTELLIGENT SYSTEMS LABORATORY PROJECT

GROUP A1-04

1.Brief explanation about the project goal

In this project we must perform a program that searchs the route between two directions of a street map.

The idea is to make this with the towns and cities of the province of Ciudad Real.

2.Tools

We choose Java as programming language to develop the project.

We have to use OSM[1] (Open Street Map) as source for the data of the cities and towns.

To manage the version control and the sharing of the project we use a repository in GitHub.

To have a better work space and to perform the work in parallel, all the members of the group use Eclipse as IDE for programming.

3.Implementation

OSM provides the graphs that we need in graphml files.

Graphml is a variation of the xml language used for the representation of graphs. These graphs contain nodes and edges.

The nodes are the different intersections of streets, and, the edges are the streets.

To read the data from the graph files we use a collection of interfaces and implementations built in Java called SAX (Simple API for XML)[2].

We use SAX because unlike other libraries for reading xml files, SAX processes the file by events, unlike the libraries that generate a hierarchical tree. Therefore it is ideal for manipulating large files, because it does not incur excessive memory usage. In addition, is simple to use.

We build 4 classes in Java (TSFGraph, Arc, Node and GraphHandler), according to the requirements of the P1 of the problem, appart of a Principal class to launch the program (P1).

- TSFGraph: We use this class to represent the Graph Structure and to implement the requested methods.
- Arc: This class represents the edges of the graphs (The streets), contains information about the nodes between which is, the name and the length.
- Node: This class represents the nodes of the graphs (The street junctions) and contains the coordinates of the node.

1: https://www.openstreetmap.org
2: http://www.saxproject.org/

- GraphHandler: This is the class that read the graphml files using the SAX library.

To save nodes and arcs we will make an adjacency list in which we will store the nodes with their adjacent nodes, which is the way in which the arcs are stored in an adjacency list.

So, we need a structure that allows us to introduce a node together with its adjacent ones, Dictionary and Hashmap provide us with keys and values.

We use a Hashmap to manage the “structure” of the graph, since the graph structure itself is very complex and it would be worse both in resources consumed and in the difficulty to develop the necessary methods.

We also think to use Dictionary, but it has been marked as obsolete by Java.

To continue with the P2, we build another 5 classes in Java:

- Frontier: This class represents the Frontier of the searching algorithm that we implement in the program. The class contains a the list of nodes that are part of the Frontier and the neccessary methods to insert and remove nodes from the same.
- State: This class represents any state that the problem can have at any moment. Contains the node in which the agorithm is, the list of nodes to those who still have to go and the md5 representation of the State.
- StateSpace: This class represents all the states that our program can adopt. Contains a list of all the states and a method to generate states for all the nodes.
- P2: This class is a representation of the problem. It’s the main class of the P2 part, performs the stress test and a demostration of the program. Also, it contains a method for check if the program reach the goal state.
- TreeNode: This class represents the node of the tree which one by the moment we only need his f value.

And we buid an extra class to read the XML files and the JSON file.

- TSFReader: This class contains the neccessary methods to read the graphml files and the JSON files, that were previously in the TSFGraph class.

We are requested to implement the program with the minimum response time as possible, so in this version, we try different data structures in the Frontier class to obtain the time results of the lists that program has to manage.

We try with 3 Java Dinamyc Data Structures, Vector, Arraylist and LinkedList.

In the P2 class you can find the `listtest` method, that measures the time consumed in add and delete elements (TreeNodes) in the 3 different DS.

We try this method 5 times with 1.000.000 elements and with 50.000 elements, and we obtain this results:

(1000000 elements)	Time in ms.					Average
ArrayList	115731	116450	120337	113411	117493	116684,4 ms
LinkedList	47	40	38	38	53	43,2 ms
Vector	118461	114352	111699	114707	123939	116631,6 ms

(50000 elements)	Time in ms.					Average
ArrayList	263	258	250	294	245	262 ms
LinkedList	14	14	13	12	12	13 ms
Vector	339	253	245	274	253	272,8 ms

We observe that the DS with the best times is the LinkedList, so, we choose it as structure to implement the list of nodes of the Frontier class.

In addition, extrapolating the results obtained, we have decided to change all the lists of the program to LinkedList, assuming that they will improve the program times.

4.Code Highlights

```java
public boolean belongNode(String id) {
    for (Node i : nodes) {
        if (i.getID().equals(id)) {
            if (adjlist.containsKey(i)) {
                return true;
            }
        }
    }
    return false;
}
```

```java
public String[] positionNode(String id) {
    String[] pos = new String[2];
    if (!belongNode(id)) {
        return null;
    } else {
        for (Node i : nodes) {
            if (i.getID().equals(id)) {
                pos[0] = i.getX();
                pos[1] = i.getY();
            }
        }
    }
    return pos;
}
```

```java
public LinkedList<Arc> adjacentNode(String id) {
    LinkedList<Node> values = new LinkedList<Node>();
    LinkedList<Arc> adjacents = new LinkedList<Arc>();
    Node aux;
    if (!belongNode(id)) {
        return null;
    } else {
        aux = returnNode(id);
        if (adjlist.containsKey(aux)) {
            values = adjlist.get(aux);
        }
    }
    adjacents = createArcs(values, aux);
    return adjacents;
}
```

This are the 3 methods requested
the first task of the project.

This is the successors method of the StateSpace class, one of the most important to the correct functioning of the program.

```
public LinkedList<Object[]> successors(State s) {
    LinkedList<Node> adj = g.adjacentNodes(s.getActualNode().getID());
    LinkedList<Node> o_list = s.getN_list();
    LinkedList<Node> n_list = o_list;
    Object[] auxReturn = new Object[3];
    LinkedList<Object[]> toReturn = new LinkedList<Object[]>();
    Arc ar = new Arc();
    State st = new State();

    for (Node a : adj) {
        n_list = o_list;
        for (Node b : n_list) {
            if (a.getID().equals(b.getID())) {
                n_list.remove(b);
            }
        }
        String a1 = "I am in " + s.getActualNode().getID() + " and I go to " + a.getID();
        st = new State(a, n_list);
        ar = g.returnArc(s.getActualNode().getID() + " " + a.getID());

        auxReturn[0] = (Object) a1;
        auxReturn[1] = (Object) st;
        auxReturn[2] = (Object) ar.getDistance();
        toReturn.add(auxReturn);
    }
    return toReturn;
}
```

With this method we insert nodes in the frontier directly placed with respect to the value f:

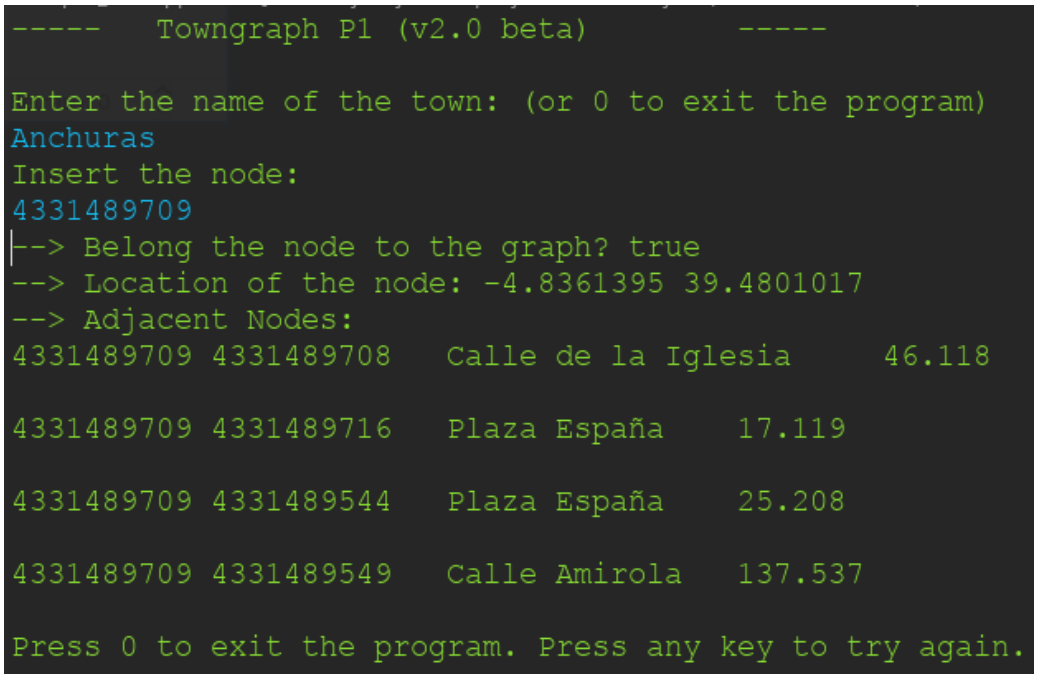
```
public void insert(TreeNode tn) {
    if (treenodes.isEmpty()) {
        treenodes.add(tn);
    } else {
        for (int i = 0; i < treenodes.size(); i++) {
            if (treenodes.get(i).getF() <= tn.getF()) {
                treenodes.add(i, tn);
                break;
            } else if (tn.getF() < treenodes.get(treenodes.size() - 1).getF()) {
                treenodes.add(tn);
            }
        }
    }
}
```

And this is the constructor of the TSFGraph class, where the most important is the way we fullfill the list with the data obtained by the `readXML(filename)` method call.

```
public TSFGraph(String filename) throws IOException, ParserConfigurationException, SAXException {
    nodes = TSFReader.parseXMLnodes(filename);
    arcs = TSFReader.parseXMLarcs(filename);
    LinkedList<Node> aux = new LinkedList<Node>();
    for (Node i : nodes) {
        aux = new LinkedList<Node>();
        for (Arc a : arcs) {
            if (i.getID().equals(a.getSource())) {
                Node n = returnNode(a.getTarget());
                aux.add(n);
            }
        }
        adjlist.put(i, aux);
    }
}
```

5.Testing

Here we show an example of the behaviour of the program with a node of *Anchuras* extracted from the document `ExampleResults` in Campus Virtual.



We also test some interval of times that the program consumes, with the example above.

- Average time of reading the *Anchuras* graphml file :
  - `30292022 ns.`
- Average time of extracting the adjacent nodes:
  - `744755 ns.`

6.Participants

This project is done by:

- Sergio Herrera Piqueras
- Juan Mena Patón
- Pablo Rodríguez Solera

7.Resources

Github page:

https://github.com/A1-04/towngraph