Group: A1-04

LAB: Problem solving and search

Sergio Herrera Piqueras
Juan Mena Patón
Pablo Rodríguez Solera

INTELLIGENT SYSTEMS LABORATORY PROJECT

GROUP A1-04

1.Brief explanation about the project goal

In this project we must perform a program that searches the route between some directions of a street map inside a city or a town.

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.

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

- Node: This class represents the nodes of the graphs (The street junctions) and contains the coordinates of the node.
- 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. It 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 a node of the tree.

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.

(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 try this method 5 times with 1.000.000 elements and with 50.000 elements, and we obtain these results: We observe that the DS with the best times is the LinkedList, so, extrapolating the results obtained, we choose it as structure to implement all the lists of the program.

However, to implement the Frontier, we have choose the DS PriorityQueue, since this DS it's very useful in order to have the frontier always order. Besides with PriorityQueue we are able to insert the nodes directly in their position for their value.

For the P3, we build only one more class on Domain and a P3 main for launch the practise:

- TSFAlgorithm: In this class are the two methods of search and the method for create a solution in case that the problem has it.
- P3: This class is the main class for the practise 3. Here the program asks the user for the data, the strategies of search and the option of prunning.

4.Code Highlights

These are the 3 methods requested in the first task of the project:

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


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;
}

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 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 = new LinkedList<>();
        Object[] auxReturn = new Object[3];
        LinkedList<Object[]> toReturn = new LinkedList<Object[]>();
        LinkedList<Node> aux_list = new LinkedList<>();
        Arc ar = new Arc();
        State st = new State();

        for (Node a : adj) {
            auxReturn = new Object[3];
            n_list = new LinkedList<>();
            aux_list = new LinkedList<>();

            // Copying the content of the parent list for security
            for (Node i : o_list) {
                n_list.add(i);
                aux_list.add(i);
            }

            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());

            // aux_list = st.getN_list();
            for (Node b : aux_list) {
                if (b.getID().equals(st.getActualNode().getID())) {
                    st.getN_list().remove(b);
                }
            }

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

With this method we compare nodes for inserting directly ordered in the Frontier:

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

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);
        }
}
```

This is the method that implements the search algorithm. It is based on the code of the method "`busqueda_acotada`" in moodle.

**This code is in another page because of his size.*

Another important part of our code is the method where we do the pruning for the problems with pruning.

```
private static void makePruning(LinkedList<TreeNode> ln, Hashtable<String,
Float> VL, Frontier fringe) {
        Iterator<TreeNode> it = ln.iterator();
        while (it.hasNext()) {
                TreeNode n_aux = it.next();
                if (!VL.containsKey(n_aux.getCurrentState().getMD5())) {
                        VL.put(n_aux.getCurrentState().getMD5(),
                                                        n_aux.getF());
                        fringe.insert(n_aux);
                } else {
                        float f_aux =
                                        VL.get(n_aux.getCurrentState().getMD5());
                        if (Math.abs(n_aux.getF()) <= Math.abs(f_aux)) {
                                VL.replace(n_aux.getCurrentState().getMD5(),
                                                        f_aux, n_aux.getF());
                                fringe.insert(n_aux);
                        }
                }
        }
}
```

```
private static LinkedList<TreeNode> bounded_search(Problem p, String strategy, int prof_max, boolean pruning, int[] n_generated)
{
		Frontier fringe = new Frontier();
		TreeNode n_inicial = new TreeNode(null, p.getI_state(), 0, 0, strategy);
		fringe.insert(n_inicial);
		boolean sol = false;
		TreeNode n_actual = new TreeNode();
		Hashtable<String, Float> VL = new Hashtable<String, Float>();
		TSFGraph g = p.getSpace().getGraph();

		while (!sol && !fringe.isEmpty()) {
			n_actual = fringe.remove();

			if (pruning) {
				if (!VL.containsKey(n_actual.getCurrentState().getMD5())) {
					VL.put(n_actual.getCurrentState().getMD5(), n_actual.getF());
				} else {
					float f_aux = VL.get(n_actual.getCurrentState().getMD5());
					if (Math.abs(n_actual.getF()) <= Math.abs(f_aux)) {
						VL.replace(n_actual.getCurrentState().getMD5(), f_aux, n_actual.getF());
					}
				}
			}
			if (p.isGoal(n_actual.getCurrentState())) {
				sol = true;
			} else {
				LinkedList<Object[]> ls = p.getSpace().successors(n_actual.getCurrentState());
				LinkedList<TreeNode> ln = makeListTreeNodes(ls, n_actual, prof_max, strategy, g, n_generated);
				if (pruning) {
					makePruning(ln, VL, fringe);
				} else {
					fringe.insertList(ln);
				}
			}
		}

		if (sol) {
			return createSolution(n_actual);
		} else {
			return new LinkedList<TreeNode>();
		}
	}
```

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.

```
-----   Towngraph P1 (v3)      -----
Enter the name of the town: (or 0 to exit the program)

Anchuras

Insert the node:

4331489709

--> Belong the node to the graph? true
--> Location of the node: -4.8361395 39.4801017
--> Adjacent Nodes:
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

Press 0 to exit the program. Press any key to try again.
```

This is an example of the output in command line of our program. We have to put the name of the Json file, the strategy to run the algorithm (you can put the acronym in Uppercase or in Lowercase), the maximum depth that you want for the strategy and if you want pruning or not.

```
-----   Towngraph P3 (v4.1)    -----

-- Insert the json filename: Anchuras

-- Select an strategy to run the algorithm --
Type the strategy (BFS, DFS, DLS, UCS, IDS, GS or A*): bfs

-- Tell me the maximum depth:
25

You type BFS. Do you want pruning? (y for yes and n for no):  y

-- You choose BFS with pruning. Running the algorithm...Maximum depth: 25--

-- Writing to an output file... --

-- Operation finished. Program closed.—
```

This is an example of what our program writes in the output.txt file that it creates after it´s execution, when the problem has a solution. We measure the generated nodes, the depth of the solution and the time that the program need to find a solution.

```
-- Solution of the algorithm:

Strategy:BFS
Generated nodes:5293
Depth:23
Cost:3583.648
Time:152 ms

1. I am in 4331431334 and I go to 4331431346   | Cost:71.161
2. I am in 4331431346 and I go to 4845150944   | Cost:124.317
3. I am in 4845150944 and I go to 4331489762   | Cost:2116.2249
4. I am in 4331489762 and I go to 4331489532   | Cost:2280.1868
5. I am in 4331489532 and I go to 946409139    | Cost:2307.6929
6. I am in 946409139 and I go to 4331489528    | Cost:2340.1748
7. I am in 4331489528 and I go to 946409156    | Cost:2538.2659
8. I am in 946409156 and I go to 4331489684    | Cost:2559.9949
9. I am in 4331489684 and I go to 4331489683   | Cost:2593.9758
10. I am in 4331489683 and I go to 4331489684  | Cost:2627.9568
11. I am in 4331489684 and I go to 946409158   | Cost:2681.7588
12. I am in 946409158 and I go to 946409159    | Cost:2714.3447
13. I am in 946409159 and I go to 4331489627   | Cost:3097.1558
14. I am in 4331489627 and I go to 4331489577  | Cost:3149.6099
15. I am in 4331489577 and I go to 4331489575  | Cost:3182.889
16. I am in 4331489575 and I go to 4331489577  | Cost:3216.168
17. I am in 4331489577 and I go to 4331489627  | Cost:3268.622
18. I am in 4331489627 and I go to 4331489570  | Cost:3291.322
19. I am in 4331489570 and I go to 4331489573  | Cost:3344.025
20. I am in 4331489573 and I go to 4331489564  | Cost:3389.4128
21. I am in 4331489564 and I go to 4331489656  | Cost:3473.4229
22. I am in 4331489656 and I go to 4331489667  | Cost:3557.1099
23. I am in 4331489667 and I go to 4762868814  | Cost:3583.648
```

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