Algorithms and Data structures
Algorithmen und Datenstrukturen

June 1, 2018

Lab 4 - Dijkstra's Algorithm

Version 1.0 - english

Submission Deadline: June 17, 2018 @ 23:59

Submission System:
`https://aud.cdc.informatik.tu-darmstadt.de/cgi-bin/index.cgi`

1 Introduction

GPS-based navigation systems allow you to find the shortest route between two points, A and B. Typically they allow you to specify routing options, such as choosing the fastest or the shortest route (which might not be the same). Using knowledge about the distances and the speed limits on the roads, the system calculates an estimated arrival time at the destination. One issue with this type of calculations is that in reality the time it takes to get from point A to B is not only based on the speed limits and the distance, but also to the number of crossings/red lights and other factors. This lab is about writing a simple navigation algorithm that takes these conditions into consideration.

2 Task

Your task is to implement a Java class (`Navigation`) that calculates the shortest path between two points on a map. The class takes as input the name of a file containing the input map. The class is to provide methods for calculating the shortest path in terms of distance or time, and to give the output either as an integer (say 25 km or 56 minutes) or as a new map where the shortest (or fastest) route is indicated.

The input map is given as a directed graph with vertices and edges. Each vertex represents a position (like a crossing), and every edge represents a road between positions. Every vertex has a name and a time (in minutes) associated the average waiting time at that particular crossing. Every edge has a speed limit (in km/h) and a distance (in km) associated with it. Your class is to output a new map where the shortest (or fastest) route is marked by making the edges on the route bold (basically thicker). For this lab the `dot` language is used for representing the map, both for input and output. See Section 4 for more information on how maps are represented.

You are to use **Dijkstra's algorithm** to find the shortest (or fastest) route from A to B. Your solution does not have to compute the shortest routes to all locations from A but only the one to B. When asked for the fastest route you can assume that the car can hold the maximum speed limit for the whole distance. Furthermore, the waiting time at the start and destination are ignored, only the intermediate waiting times are considered. Some vertices may have a waiting time of 0 minutes, indicating that they are not crossings, but other locations of interest. When calculating the distance (or time) needed, use the built-in Java type `double`. When returning an answer, return it as an integer (`int`), rounded upwards (thus both 5.25 and 5.75 become 6). For this you can use the method `Math.ceil`.

3 The Code

We provide the following Java classes and test files within the zip archive **Lab4.zip** which contains the Java project with the building blocks. The classes are split in two packages– the 'frame' and the 'lab' package. Additionally, we provide the correct output maps related to the test cases and the provided test files in **Lab4Output.zip**.You should implement your solution starting with the skeleton code provided in the zip archive.

3.1 The 'frame' package

The submission system will use its own copy of these classes when testing your source code. Changes made by you within this package will not be considered by the submission system.

3.1.1 `AllTests.java`

This is the JUnit test case class. The test cases defined by this class are used to test the correctness of your solution. This class is also responsible for writing the output maps into the project directory.

3.2 The 'lab' package

This package contains the files you are allowed to modify. You are free to add additional classes (in new files within the lab package), as well as to add any additional methods or variables to the provided classes, as long as you **do not change the signature (*name, parameter type, and number*) of any given method**, as they will be used by the JUnit tests. Any source code changes that you make outside the 'lab' package will be ignored when you upload your source code to the submission system.

3.2.1 `Navigation.java`

For the `Navigation` class the following methods are to be implemented:

- `public Navigation(String filename)`

  The constructor of the class takes the name of the file containing the map as input. You can assume that the file is located in the same directory as the executable program.

- `public ArrayList<String> findShortestRoute(String A, String B)`

  This method returns the shortest route (the route with the *shortest distance*) from point A to point B. It returns a `ArrayList<String>` containing the output map. Each edge on the shortest path from A to B should be marked bold as described in Section 4. Each item in the `ArrayList` corresponds to one line of the output map. The output should be directly interpretable dot code. For example, the for the map depicted in Figure 6, the output ArrayList would be of length 20. The

two points are identified by their names as given in the input file. The contents of the input file is not to be changed.

If one or both points are not on the map or if there is no path from A to B the returned map should be the original map, without marked edges.

The order in which the edges appear in the output map need not be identical to the input map. However, all edges must occur somewhere in the output map.

- `public ArrayList<String> findFastestRoute(String A, String B)`

  This method works exactly like `findShortestRoute` but finds the *fastest* route instead. The output is again to be the input map with the edges on the fastest route marked bold.

- `public int findShortestDistance(String A, String B)`

  This method returns the shortest distance in kilometers between points A and B. If point A is not on the map it returns -1, if point B is not on the map it returns -2, if both are not on the map it returns -3. If there is no path between point A and point B it returns -4. If point A is identical to point B it should obviously return 0.

- `public int findFastestTime(String A, String B)`

  This method returns the shortest time in minutes between points A and B. If point A is not on the map it returns -1, if point B is not on the map it returns -2, if both are not on the map it returns -3. If there is no path between point A and point B it returns -4. If point A is identical to point B it should return 0.

3.3 Test Files

We provide two input files for testing as follows:

- `Testfile1` and
- `Testfile2`.

You are encouraged to test your solution using additional input files as well. Note that you need to write your own JUnit test cases in order to run with your customized input graphs. Think about the assumptions made on the input. To make sure that your solution works, do test it with all the test cases provided. Apart from the given input files, the submission system will test your solution with several additional input files, to confirm the correctness of your program. Those additional input files are made available to you but are not included in the zip file.

3.4 Additional Hints

- You may start with implementing a class to represent edges and one to represent vertices, in order to finally represent the graph as a list of edges and a list of vertices.
- Use the constructor of the `Navigation` class to read the test file into your graph representation.

4 In- and Output Format: dot

As input and output format for this lab we will use a subset of the `dot` language for directed graphs. This section will only describe this subset. Please refer to the online documentation for getting more information and downloading tools:

`https://www.graphviz.org`

4.1 dot Language

Each file starts with a line containing the word `digraph` indicating that we are working with directed graphs. Next follows a "{". The file ends with the corresponding "}". Vertices are implicitly defined by defining the edges between them. For instance, the two vertices A and B are connected by an edge. The definition of the simple graph in Figure 2 is given by the following three lines:

```
digraph {
A -> B;
}
```

Figure 1: The code for Figure 2

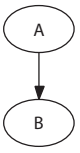

Figure 2: A simple directed graph

One can change the properties of an edge (and of course also of vertices) by adding markups. We will only use two types of markups, namely making an edge thicker and adding a label to it. This markup is added using brackets, for instance a thicker edge with the label `10,110` is added like this:

```
digraph {
A -> B [label="10,110"][style=bold];
}
```

Figure 3: The code for Figure 4

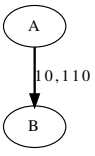

Figure 4: A simple directed graph with bold edge and label

We will use the labels to represent both the distance (in km) between two vertices (length of an edge) and the speed limit (in km/h) as `"Distance,Speed limit"`. Bold edges represent an edge that is on the shortest path between two vertices. For the vertex labels we will use `"Name,Time"` to represent the name of the vertex and the average waiting time (in minutes) as can be seen in the example given in Figure 5.

4.2 Example

The graph in Figure 6 and corresponding code is one of the inputs to the lab. We added line numbers to the code in Figure 5.

```
1.  digraph {
2.  A -> B [label="10,90"];
3.  A -> C [label="8,80"];
4.  C -> B [label="1,50"];
5.  B -> D [label="7,60"];
6.  C -> D [label="6,80"];
7.  D -> E [label="4,90"];
8.  E -> F [label="2,130"];
9.  D -> F [label="5,130"];
10. F -> G [label="5,120"];
11. G -> H [label="5,100"];
12. A [label="A,5"];
13. B [label="B,4"];
14. C [label="C,3"];
15. D [label="D,2"];
16. E [label="E,1"];
17. F [label="F,6"];
18. G [label="G,7"];
19. H [label="H,8"];
20. }
```

Figure 5: Example code for an input map in the dot language

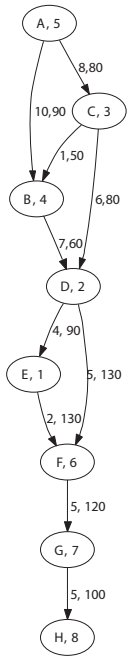

Figure 6: A simple directed graph from the code in Figure 5

You can assume that all the input maps will follow the format in Figure 2, i.e., first containing the edge definitions with the labels, followed by the vertex label definitions. The name of a vertex written in the label will always be the same as the vertex name in `dot` (the vertex `A` on line 2 is always called `A` in the vertex label definition, here on line 12).

4.3 Producing Graphs

`dot` can be used to export to many different formats, such as PostScript, gif, jpeg, png etc. You can download the latest version from the project web page and run it on your computer.