The University of Newcastle
School of Electrical Engineering and Computer Science

COMP2230/6230 Algorithms

Assignment

Total mark: 100
Due 26 October 2018, 11:59pm in the Blackboard

TASK

1. Design and implement an efficient algorithm for determining the best route between two given stations in a given rail network.

2. The rail network should be passed to your program as an input. We will use a simplified version of the Sydney CityRail network given to you in RailNetwork.xml

3. Your program should NOT take into account the time of the day, but rather use an average time to travel from a station to a station on a particular line, and a flat time of 15min needed to change from one line to another. This information will be given as a part of the input rail network data.

4. Your task is to optimise the route based on the time traveled.

This assignment can be done individually, or in pairs. If the assignment is done in pairs, then you there will be an extra task:

5. **(Extra: for pairs)** Allow optimisation of the route based on the number of 'changes' and the time travelled, in the order selected by the user. Here, 'change' refers to a change from one rail line to another.

You are encouraged to do the assignment in pairs.

Your program should contain a header with information about what it does and what the input and output are. You must use a version of Dijkstra's algorithm that uses adjacency lists and indirect heaps.

Your program should also contain inline comments and be easy to follow.

The programs should be written in Java or C++.

INPUT

Your program must be able to be used from the command line and must accept *all* input as command line arguments.

1. Your program must take as an input the name (including path if necessary) of an XML file containing an undirected graph of the rail network:

- The graph is provided as a list of neighbours for each vertex in the graph.
- Each vertex corresponds to a station in the network, and each vertex name consists of 2 strings: The name of the station, and the name of the rail line. A single station might thus be represented by multiple vertices if it is on multiple lines.
- Each entry in the list of neighbours of vertex *v* contains the name of the neighbouring vertex, say *u*, followed by the weight of the edge *vu*. If the vertices v and u are on the same line, the weight of the edge *vu* is the time in minutes needed to travel from vertex *v* to vertex *u* (or vice-versa – it is assumed that the time is the same for both directions). If the vertices *v* and *u* are not on the same line, the weight of the edge *vu* is 15 (we are assuming that it takes 15 minutes to change lines at a station).

A sample input file *RailNetwork.xml* will be provided in Blackboard. You don't need to create other input files – you can use this one for the purpose of developing and testing your assignment.

It is advised that you investigate the many pre-existing libraries (for Java or C++) available for working with XML and incorporate those into your program. Then, reading the data from the xml file and navigating the data should amount to (reasonably) simple calls to the classes and methods of the library in question.

2. Your program must accept as input the names of two stations, say X and Y.

3. Pair Assignment: In addition to the above, your program must accept as input the optimisation criterion. This must be `time` for minimum total time traveled, or `changes` for minimum number of 'changes'. If there are multiple optimal results satisfying the chosen criterion, your program must output the one that optimises the other criterion.

The order of command line arguments must be:

xml file, station 1, station 2, optimisation criterion (if appropriate)

see SUBMISSION section, below for more information.

OUTPUT

1. Individual assignment:
   You program must output the quickest route in the following format:

```
From X, take line a to station Z;
then change to line b, and continue to W;
…
then change to line c, and continue to Y.
The total trip will take approximately n minutes.
```

2. Pair Assignment:
   You program must output the quickest route in one of the following formats, depending on the selected optimisation criterion:

```
From X, take line a to station Z;
then change to line b, and continue to W;
...
then change to line c, and continue to Y.
The total trip will have m changes and will take approximately n
minutes.
```

   OR

```
From X, take line a to station Z;
then change to line b, and continue to W;
...
then change to line c, and continue to Y.
The total trip will take approximately n minutes and will have m
changes.
```

SUBMISSION

Your submission should consist of the code, and a readme file containing any special instructions for your program. Your main function must be named assign1 (or assign1.java)

Your code must compile and run using command line statements, such as:
Java:

```
javac assign1.java
java assign1 "xml_file" "station 1" "station 2" criterion > output1.txt
```

C++:

```
g++ -o assign1 *.cpp
./assign1 "xml_file" "station 1" "station 2" criterion > output1.txt
```

- The first argument must accept a *path* to the file (so that files in other folders can be used).
- The station names must only be the name of the station (e.g. "Kings Cross"), with no line information.
- The criterion argument is only needed for the pair assignment.
- Output must be to standard out, so that it can be redirected as in the above examples.

ASSESSMENT CRITERIA

Individual Assignment:

1. 20 marks for correctly reading the input graph;
2. 30 marks for the correct Dijkstra's algorithm;
3. 30 marks for an efficient implementation of Dijakstra's algorithm using adjacency lists and indirect min-heap;
4. 10 marks for a clean and well-commented implementation;
5. 10 marks for the correct output format and result.
6. -5 marks *for each time* the marker must modify the submission to make it work.
7. -2 marks *for each* requirement of the program not met (e.g., name of executable, correct order of command line arguments, etc)

Pair Assignment:

1. 18 marks for correctly reading the input graph;
2. 28 marks for the correct Dijkstra's algorithm;
3. 28 marks for an efficient implementation of Dijakstra's algorithm using adjacency lists and indirect min-heap;
4. 8 marks for a clean and well-commented simple mentation;
5. 8 marks for the correct output format and result;
6. 10 marks for two-criteria optimisation.
7. -5 marks *for each time* the marker must modify the submission to make it work.
8. -2 marks *for each* requirement of the program not met (e.g., name of executable, correct order of command line arguments, etc)

Note that modification of a submission will be considered *only* in order to make the program compile or run so that it may be tested. Modifications will only be performed if they are extremely easy and quick to do so.