Transportation Network Documentation

Ormir Gjurgjej, Nitika Kumar

June 3, 2016

1 Introduction

The first thing to consider when working with graphs is what type of graph you are working with. Here we have the Viennese public transportation network that clearly consists of more vertices (stations) than edges (connection between each vertex) which is why the implementation of an adjacency list has been more beneficial in our case rather than the use of an adjacency matrix.
The aim of this program is to find the shortest path from a starting point to a destination using the Priority First Search as in Dijkstra's algorithm. The purpose of this documentation is to explain each step taken for the implementation of this algorithm making it possible for any user to calculate an overall travel time in order to reach a location from any starting point whereas both - start and finish - are completely arbitrary.
We started off by creating two classes - Network and Station. Each object of the second class which represents one station of the transportation network system is the vertex of the graph. The minutes inbetween stand for the edges that connect every vertex.

2 Introducing the Datastructures

2.1 Station

This class is responsible for the vertices of the entire graph. As the adjacency list requires us to do, each vertex contains the name of the station, the line it belongs to (either metro or tram) and a map which contains a reference to the neighbours (nodes) next to the current vertex that we are positioned on including the distance as in how long it takes to get there. The following code snippet shows the private members of the Station class :

```
class Station {
    std::string name;
    std::string line;
    std::map<Station*, int> neighbours;
    ...
};
```

Listing 1: class Station(private members)

Furthermore this class contains several functions, a constructor and a destructor.
One of them is getDistance(). The distance from one vertex x to another vertex y can be defined as :

$$d(x,y) = \begin{cases} \text{Min } \{\text{w(p)} \mid \text{p is the way from x to y}\} \text{ if a way exists} \\ \text{Infinite if no way exists} \end{cases}$$

The infinite in our case is defined in the header file station.hpp as INF with the value 999 which means that if the name of the station and its distance are not found anywhere in the "neighbours" map, this function returns INF.

```
int Station::getDistance(Station* nStation) const {
    try {
        return neighbours.at(nStation);
    } catch (std::out_of_range e) {
        return INF;
    }
}
```

Listing 2: getDistance()

The info() function returns a beautiful output of the entire tranportation network that was imported by the user in the terminal. Each station is listed in an alphabetical order, here is a snippet :

Spittelau (U4, U6, D)
    2 min to Heiligenstadt (U4)
    1 min to Friedensbruecke (U4, 33)
    1 min to Jaegerstrasse (U6, 31, 33)
    2 min to Nussdorfer Strasse (U6, 37, 38)
    1 min to Liechtenwerder Platz (D)
    2 min to Rampengasse (D)
Stadion (U2)
    1 min to Krieau (U2)
    2 min to Donaumarina (U2)
Stadiongasse/Parlament (1, 71, D)
    2 min to Dr.-Karl-Renner-Ring (1, 2, 46, 49, 71, D)
    1 min to Rathausplatz/Burgtheater (1, 71, D)
Stadlau (U2)
    2 min to Donaustadtbruecke (U2)
    1 min to Hardeggasse (U2, 25)
Stadlergasse (10, 60)
    2 min to Gloriettegasse (10, 60)
    2 min to Jagdschlossgasse (10, 60)
Stadtpark (U4)
    2 min to Karlsplatz (U1, U2, U4, 1)
    1 min to Landstrasse (U3, U4)
Staglgasse (52, 58)
    1 min to Gerstnerstrasse/Westbhf. (52, 58)
    2 min to Kranzgasse (52, 58)
Stammersdorf (30, 31)
    2 min to Van-Swieten-Kaserne (30, 31)
Stefan-Fadinger-Platz (1)
    1 min to Windtenstrasse (1)
Stephansplatz (U1, U3)
    1 min to Schwedenplatz (U1, U4, 1, 2)
    2 min to Karlsplatz (U1, U2, U4, 1)
    1 min to Stubentor (U3, 2)
    2 min to Herrengasse (U3)

```
void Station::info() {
    std::cout << name << " (" << line << ")" << std::endl;
    for (auto it = neighbours.begin(); it != neighbours.end();
   ++it) {
        std::cout << '\t' << it->second << " min to " << it->
    first->getName() << " (" << it->first->getLine() << ")" <<
    std::endl;
    }
}
```

Listing 3: info() displays the network

2.2 Network

Network.cpp consists of functions that enable the user to ask for help (opening the README.txt file), to import a file of a transportation network by typing "import", to show all the stations and lines thanks to the info() function as well as executing the dijkstra algorithm by typing in "search" followed by the starting point and destination (calling the path() function).

3 Dijkstra's Algorithm

Perhaps the most difficult part of this project was the implementation of Dijkstra's algorithm. The reason why we used this algorithm and not Bellman-Ford's one is because the edges do not have a negative weight. It all takes place in Network's function called path() which takes two parameters - the name of the starting point and the name of the destination. You will find the entire code snippet below.
First, we ensure that the stations entered by the user are part of the network (ubahn.txt). If so, we move on to cmp that stores a condition that we will need for the priority queue afterwards. We used "pair" as a datatype as it stores two values : a pointer pointing to the neighbour nodes (Station*) and the amount of time it takes to reach this station (int).
The good thing about priority queues as opposed to normal queues is that you can have a certain criterion for the order of the elements in the queue, which is defined in the "decltype(cmp)>queue(cmp);". This means that whenever the condition left.first>right.first (distance of left vertex greater than the

distance of the right vertex) is true, add this element by first comparing it with each element in the priority queue and then pushing it in at the right place.
In order to keep an overview of the stations that were visited we used a <set>to store all the stations on the way from source to destination. We also created a <map>where we save the minutes/travel time each time we iterate over the vertices.
Here comes the defining part : head is a pointer always pointing to the top element of the queue (as we used a priority queue, mentioned previously) or in other words, it is always the current position that we are located on between source and destination. Every time head moves over to the next station, the top element is first removed with the queue.pop() function. If head is not part of the visited stations (set) yet, insert it into the set. This process is repeated as long as head finishes iterating over every single element in the queue.
We then create another map called nbList and assign all the neighbours of head to it. Eventually, we need to iterate over each neighbour station and just check if the distance of the vertex has already been saved in the distance map. If the new distance (newDis) is shorter than the older distance (dist[nb->first]), it is added to the map dist. newDis contains the distance from source to the current station and in addition to the neighbour. dist[nb->first] however just stores the distance between source and any neighbour station. Finally, the distance variable is updated and adds together all the minutes it found along the shortest path between source and destination.
In order to display the path we iterate over the neighbours again but in reverse.

```
1  bool Network::path(std::string bName, std::string eName) {
2          Station *source, *destination;
3      try {
4          source = stations.at(bName);
5          destination = stations.at(eName);
6      } catch (std::out_of_range e) {
7          std::cout << "Couldn't find begin or end station" << std
       ::endl;
8          return false;
9      }
10
11     Station * head = source;
```

```
12
13      auto cmp = [](std::pair<int, Station*> left, std::pair<int,
    Station*> right) {
14          return left.first > right.first;
15      };
16
17      std::priority_queue<std::pair<int, Station*>, std::vector<
    std::pair<int, Station*>>,decltype(cmp)> queue(cmp);
18      queue.push(std::make_pair(0, source));
19
20      std::set<Station*> visited;
21      std::map<Station*, int> dist;
22      dist[source] = 0;
23
24      while (!queue.empty()) {
25          head = queue.top().second;
26          if (head == destination) break;
27          queue.pop();
28          if (visited.find(head) != visited.end()) continue;
29          visited.insert(head);
30
31          std::map<Station*, int> nbList = head->getNeighbours();
32          for (auto nb = nbList.begin(); nb != nbList.end(); ++nb)
    {
33
34              auto dis = dist.find(nb->first);
35
36              int newDis = dist[head] + head->getDistance(nb->
    first);
37              if (dis == dist.end() || newDis < dist[nb->first]) {
38                      dist[nb->first] = newDis;
39                      queue.push(std::make_pair(head->getDistance(
    nb->first), nb->first));
40              }
41          }
42      }
43
44      std::cout << "It takes " << dist[head] << " min from " <<
    source->getName() << " to " << destination->getName() << std
    ::endl;
45      std::string wayyy = head->getName();
46      int distance = 0;
47      while (head != source) {
48          Station * nbShort = nullptr;
49
```

```
50          std::map<Station*, int> nbList = head->getNeighbours();
51          for (auto nb = nbList.begin(); nb != nbList.end(); ++nb)
        {
52              if (dist.find(nb->first) == dist.end()) continue;
53              else if (nbShort == nullptr) nbShort = nb->first;
54              else if (dist[nbShort] > dist[nb->first]) nbShort =
    nb->first;
55          }
56
57          if(std::prev(nbShort)->getLine() != std::next(nbShort)->
    getLine()){
58              dist[nbShort] = dist[nbShort] + 5;
59          }
60
61          wayyy = nbShort->getName()+ " -> " + wayyy;
62          head = nbShort;
63          distance = dist[nbShort];
64      }
65
66      std::cout << wayyy << distance << std::endl;
67
68      return false;
69  }
```

Listing 4: Dijkstra's algorithm implemented in path()

4 Test Protocol

4.1 Michelbeuern-AKH to Dresdner Strasse

search
Choose your starting point:
Dresdner Strasse
Choose your destination:
Michelbeuern-AKH
It takes 7 min from Dresdner Strasse to Michelbeuern-AKH
Dresdner Strasse ->Jaegerstrasse ->Spittelau ->Nussdorfer Strasse ->Waehringer Strasse-Volksoper ->Michelbeuern-AKH

According to "Wiener Linien" it does indeed take 7 min to go from Dresdner Strasse to Michelbeuern-AKH.

4.2 Hietzing to Schottentor

search
Choose your starting point:
Hietzing
Choose your destination:
Schottentor
It takes 16 min from Hietzing to Schottentor
Hietzing ->Schoenbrunn ->Meidling Hauptstrasse ->Laengenfeldgasse ->Margaretenguertel ->Pilgramgasse ->Kettenbrueckengasse ->Karlsplatz ->Museumsquartier ->Volkstheater ->Rathaus ->Schottentor

According to "Wiener Linien" it takes 19 min from Hietzing to Schottentor due to modernisation work at the station "Hietzing". Normally I suppose it wouldn't take as long.

4.3 Spitalgasse/Waehringerstrasse to Karlsplatz

search
Choose your starting point:
Spitalgasse/Waehringer strasse
Choose your destination:
Karlsplatz
It takes 13 min from Spitalgasse/Waehringer strasse to Karlsplatz
Spitalgasse/Waehringer strasse ->Waehringer Strasse-Volksoper ->Nussdorfer Strasse ->Spittelau ->Friedensbruecke ->Rossauerlaende ->Schottenring ->Schwedenplatz ->Stephansplatz ->Karlsplatz

According to "Wiener Linien" it does indeed take at least 14 min from Spitalgasse/Waehringerstrasse to Karlsplatz.

4.4 Gersthof to Friedrich-Engels-Platz

search
Choose your starting point:
Gersthof
Choose your destination:
Friedrich-Engels-Platz

It takes 15 min from Gersthof to Friedrich-Engels-Platz
Gersthof ->Simonygasse ->Sommarugagasse ->Vinzenzgasse ->Hildebrandgasse ->Michelbeuern-AKH ->Waehringer Strasse-Volksoper ->Nussdorfer Strasse ->Spittelau ->Jaegerstrasse ->Hoechstaedtplatz ->Friedrich-Engels-Platz
According to "Wiener Linien" it takes at least 19 min from Gersthoff to Friedrich-Engels-Platz.

Please bear in mind that the edges (travel time in min) given in the ubahn.txt file might not be entirely accurate leading to some mismatches in the timings.