Algorithmy Project

SWOLE MICHELIN

Realised by Nicolas DEVIERS

This project follows the previous Michelin project, whose point was to encounter the shortest route between two metro stations in Santiago's network.
The second version allows the user to find the shortest itinerary to visit several stations, starting from a specific one.

The algorithms used in this project are mainly all of my own making, I however Stack Overflowed a bit in order to produce my program's efficiency.

Explanation of the first program's inner functioning, which is embedded is this software.

In order to do this, a graph of the whole network is generated the following way : each metro line is interpreted as a doubly linked list, and then at each line intersections the nodes are merged in order to connect both lists. Therefore each node can be connected to a maximum of 4 other nodes, representing two metro lines crossing each others' path.

To start the program, the user has to give it as argument the name of the two stations he is departing from and going to. The program therefore has to be executed on a terminal or with a bash script.

Once the graph has been generated, the program will start from the destination point and spread everywhere, giving to every node a weight proportional to it's distance from the destination. So the destination's weight will be 1, and a station 5 node away will have a weight of 6 for example.
Following this logic, the weight of a node defines it's distance from the destination point, and the linked node with an inferior weight shows the way towards the destination.

For obvious reasons the executable is not in this archive. You can generate it with the Makefile, using the simple command "make a" on any linux distribution.
This project was realized with g++ 7.2.0 on Ubuntu 17.10.

The weight assignation of the nodes is handled by the following simplified algorithm , which can be found on the PathFinder.cpp file of the project :

```
Exploration (node, distance)
{
      node.weight = distance
      While node has neighbours
            If neighbour has no weight yet, or is heavier than than this node
                  Exploration(neighbour, distance + 1)

}
```

After that, the second algorithm finds back the shortest way from the start to the exit following the decreasing weights, the following way :

```
DoYouKnowDaWey (entry)
{
      For entry's weight iterations
            While neighbor's weight != entry's weight -1
                  get next neighbor
            entry = next neighbor
            print entry's new name /* it is the next node towards the exit */
}
```

In the end, the program works fine and finds the route almost instantly.

Explanation of the Swole Michelin's inner functioning, using the Michelin.

The stations are given in argument or the program, which has to be executed like the previous version, with the slight difference that the arguments will be listed as such : departing station, number of stations of interest, name of each of those stations.

Once the map of the metro network is generated by the Michelin, this software will find the shortest route between each station the user is interested in visiting, and store those in a square matrix

This part includes an algorithm that is not of my own making : the travelLength() function in the Core object will organize and order the calculus of the distance between each stations of interest. Found on Stack Overflow.

Once this matrix is formed a recursive Travelling Salesman algorithm is applied to it

```
travelling(graph, visited, currPos, size, count, cost, ans)
{
    if there is no more station to visit
        if this path's weight is inferior to the previous one
            save the path
        store this weight
        return
    for the number of stations to visit
        if a station has not been visited
            Mark it as being visitated
            Store it's ID
            Travel again from this one station
            Unmark it as beeing visitated
            Unstore it's ID
};
```

The algorithm was inspired by those article :
https://www.hackerearth.com/practice/algorithms/graphs/hamiltonian-path/tutorial/
https://www.geeksforgeeks.org/travelling-salesman-problem-implementation-using-backtracking/

Once the best path has been encountered, the id of the stations in order is crossed with the name of the stations, then with the name of every casual station between each one of them as recuperated by the Michelin, which are printed in order.

In the end, the program works fine and finds the route almost instantly. It has no Valgrind error, however loses around 900 bits at each execution.

For obvious reasons the executable is not in this archive. You can generate it with the Makefile, using the simple command “make a” on any linux distribution.
This project was realized with g++ 7.2.0 on Ubuntu 17.10.