Faculty of Engineering
of the University of Porto

Report Part II

Algorithm Design and Analysis
Master in Informatics and Computing Engineering

Multi-modal Routing For Collective Transportation

Authors:	*Student Number:*
Afonso Jorge Ramos	up201506239@fe.up.pt
André Ferreira da Cruz	up201503776@fe.up.pt
Edgar Gomes Carneiro	up201503784@fe.up.pt

10th April 2017

Contents

1 Introduction

In the domain of the course unit of "Algorithms Design and Analysis" of the Master in Informatics and Computing Engineering course, we were expected to implement, using C++, a navigation system that took advantage of the multi-modality of nowadays' public transportation. Henceforth, this multi-modality is what enables us to alternate between the several means of public transportation in the same journey which are often sought in parallel by users, for example, the subway, bus, urban trains, etc., which is one of the features that generates vast attractiveness for the current public transportation methods.

In a nutshell, using our implementation of algorithms developed along the semester we came with a solution that we find efficient, scalable and that fits its purpose. Through this report we will demonstrate the several steps of the development of this project along with some self-criticism to make us aware of our problems and better oneself. The main idea of this report is to elaborate on pathfinding algorithms and praise its main concepts.

2 Presented Problem

At first sight, it may seem like a forthright problem to solve, however while developing this application we found that while being a simple project, it can bring several hard implementations afloat. In order to reach the end goal, we had to implement a navigational system that would be able to generate several route options between the source and the end of the journey while considering the combination of the several methods available in specific sections of the itinerary. Therefore, allowing a user to choose to go through an explicit segment while preferring a certain transportation method over the others. Furthermore, we also must take into consideration the time and cost of each possibilities while letting the user define a cost, time and maximum walking limit, thus we also need to be aware if the limitation doesn't go under the minimum calculated value.

To address the problem of finding an optimal path between two points of an inter-modal transportation network we are suggested to use an multi-modal transportation network. This network involves a set of points, also called nodes, and a set of links between pairs of nodes, usually called edges. This kind of network is generally abstracted by the mathematical concept of graph due to its similarity. Each node in the network represents a point of connection between roads or, an intersection or, a station or a public transportation stop, and it has an associated position in the xOy-plane, and an average wait time. Each route is either associated with a given transportation, like bus or subway, or mean that the user is walking from one point to the other, which can either be to switch transportation or just walk as a mean of transportation. Each of these edges (node to node), have an overall average travel time, both of which may be considered as parameters for calculating the cost of traversing the network.

In addition to the previously proposed features, we are expected to consider the several stops (bus and subway) and street names, in order to implement a search feature, both approximate and exact, so that users have a way of verifying the existence of a certain stop or street. For the exact search, if the search term is not found it must return a message indicating "unknown place", while for the approximate search it must return the stops which have similar names ordered by how similar they are.

3 Implemented Solution

3.1 Input

To test our algorithm and overall structure of the system we use as an input at least five different files. We start by selecting a section of the map using `openstreetmap.org` then we extract the information present in that section and run it through the provided parser that creates an easily readable file that supposedly only includes the information that we do need. However, it also eliminates some very important data that we do need, for example, the location of the subway stations and bus stops in spite of it being present in the initial file extracted from the website. For that reason, we still have to, manually create another file for each transportation method so that the graph knows where the bus or the subway can travel through.

3.2 Expected Output

The expected output of this system is a sequence of nodes, equivalent to the several edges between each one of them, which correspond to an optimal path in the network. However, there are numerous criteria to choose this optimal path in the grid. Our system deals with three examples of this criteria, the ones that were mentioned in the objectives for our theme, which are the path which represents the fastest journey, a path which takes into consideration his favourite mean of transportation, and the cheapest path to take. Note that to implement the price of these trips we needed to attribute a fixed price, one for the bus and one for the subway, for each node-to-node trip, but always taking into consideration the maximum walking time. Because of this, there are three possible definitions for the objective function.

3.3 Objective Function

As mentioned before, the algorithm deals with three objective functions, the ones that were suggested in the objectives for our project theme, which are a path that minimizes the time wasted in the journey, a path that prefers a mean of transportation over the others, and one that limits the cost for the user, which could be easier if it was just minimizing it, but since we are putting just a preferred cost, the algorithm has to do a lot more work than it would otherwise.

4 Implementation

Based on the ideas and concepts presented to us in classes we started by engineering a list of requirements, functionalities that we wanted to implement right away and objective goals for the final state of the project.

We then proceeded to choose a map section that would be a good candidate for testing the algorithm, since it could neither be too large of an area so that it would not affect too much our testing times and nor be too small because, to fully test the algorithm it must provide several different situations including bus, subway and walking, while encapsulating at least some subway stations. However, it also has to be easy to read by the user, since packing too much information in a small space is always a poor choice. So we chose an area near our Faculty that included both subway and bus lines and an area large enough to put the implementation to the test.

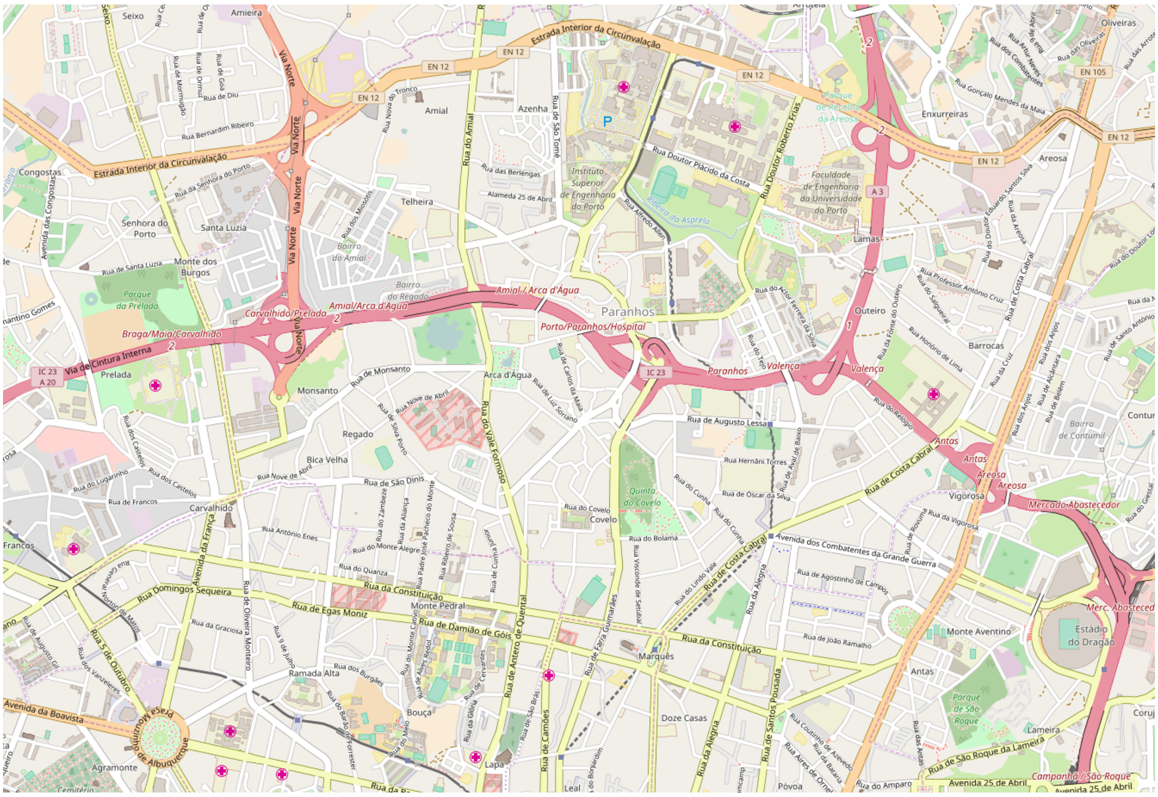
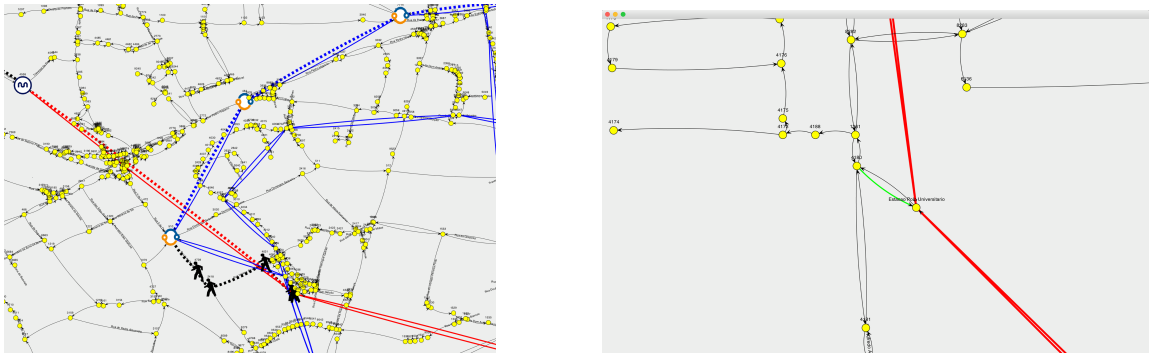


Figure 1: Map of the selected area.

To comply with these objective functions, in addition to the "basic" obstacles we discussed up until this point, we still have some big steps ahead, since, for example, not every street goes both ways, which means edges can have a specific direction. However, this was easily achieved just by creating an edge for both directions to fix the "two-way street problem". Furthermore, we must take into consideration that a user can travel by several methods with different travel times, since each method has its own speed. For example, a subway does not travel at the same speed as a man walks, and, for that reason, we essentially attribute a specific average speed to each transportation method, which when calculated along with each edge length we obtain the travel time which is used to calculate the edge weight. Which means that another point of complexity is added to the graph by defining what can go through a certain edge or not since not every edge is a bus stop or subway station. We also take into consideration the medium waiting time of the public transportation as well the cost of ticket for entering in it (prior to this implementation we calculated the price of the trip based on distance, but in order for it to be as close to reality as possible we chose this approach).

In terms of the visual appeal of the graph viewer we tried to diversify to make everything more appealing and user-friendly. We started by making public transport lines displayed with wider edges to distinguish them from the others, while painting the bus lines blue and subway ones red. The nodes which are included in these public transportation lines are also changed to suit each kind of stop, for example, bus stops have the *STCP* logo representing the node, while the subway has the *Metro do Porto* logo; both of these have the stations names displayed on the map. Repeated road names are displayed only once, to maintain map readability and to avoid packing up too much unnecessary information to the user. The edges containing the optimum path for the configuration requested by the user are then transformed into a wider and dashed line instead of a straight one as it is before being the selected path. The nodes that the user passes by on foot are changed to an image of a walking stick man.



(a) Example of the Graph Viewer interface. (b) Station to road connection.

Figure 2: Graph Viewer and connection edge.

Consequently, we made a series of changes to the traditional Graph/Edge/Vertex implementation: we calculated each edge's length from the vertices' geographical coordinates, and assigned a specific mean of transportation to each edge, which is used when calculating the edge's weight for the objective function. If the chosen objective function was the transportation preference, the path presented as the optimal one does not blindly choose that specific mean of transportation. What it does is analyse the whole path and compare if it is worth sacrificing the main objective function (least amount of time spent) for the sake of this preference, according to an adjustable setting. This setting can regulate the strength of the preference varying from weak, medium, or strong preference, while the latter always chooses the vehicle when possible. The other change that we proceed to apply was the average wait time which was achieved by defining a connecting edge which connects the station with the closest road node that we defined that took 3 minutes (based on average wait times of real-life situations).

(a) Weak preference of bus. (b) Strong preference of bus.

Figure 3: Examples of preference bus strength.

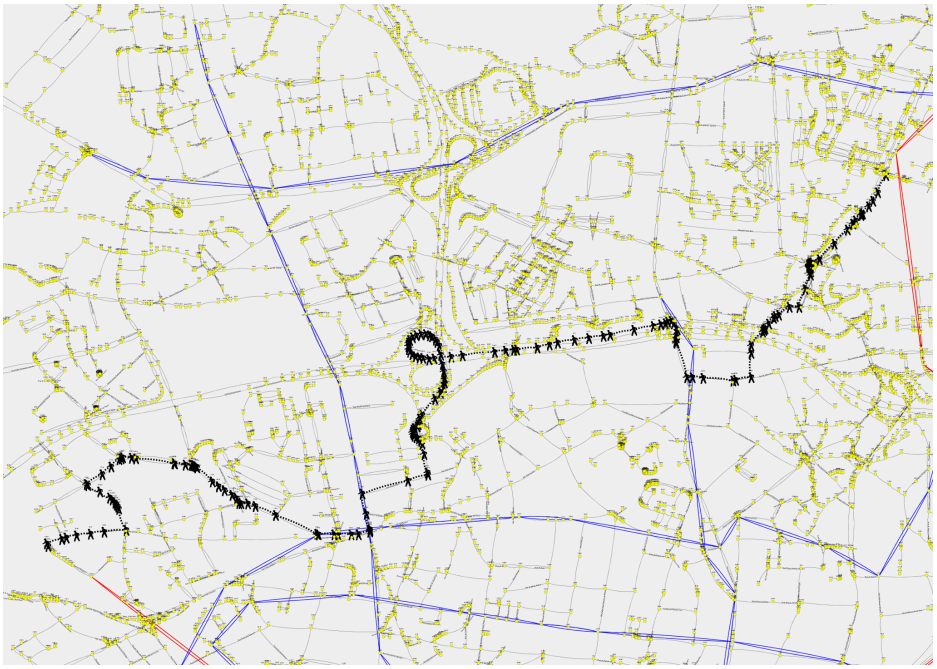


Figure 4: Example of strong preference of walking.

With the increase of requirements of the second part of this project, we decided to start with the extensive testing of the search algorithms that we were proposed to use. After polishing the algorithms to suit our needs, the exact search and the approximate search, which searched for specific word construction and similar word construction respectively, we then proceeded to implement these features within our program. With the growth of the number of functions present in our implementation we decided to add a menu so it would be easier to navigate through the several functions.

```
** Main Menu **
   1 - View Graph
   2 - Ask for path
   3 - Exact string matching
   4 - Approximate string matching
   5 - Exit program
```

(a) Menu.

```
** Approximate Match **
Transport stop's name: Faria

Matches per proximity (w/ max. distance 3) :
0 -- Estacao Faria Guimaraes
1 -- Hospital Sta. Maria
3 -- Arca d'Agua


Press any key to continue...
```

(b) Approximate Search.

Figure 5: Menu and Search.

5 Class Diagram

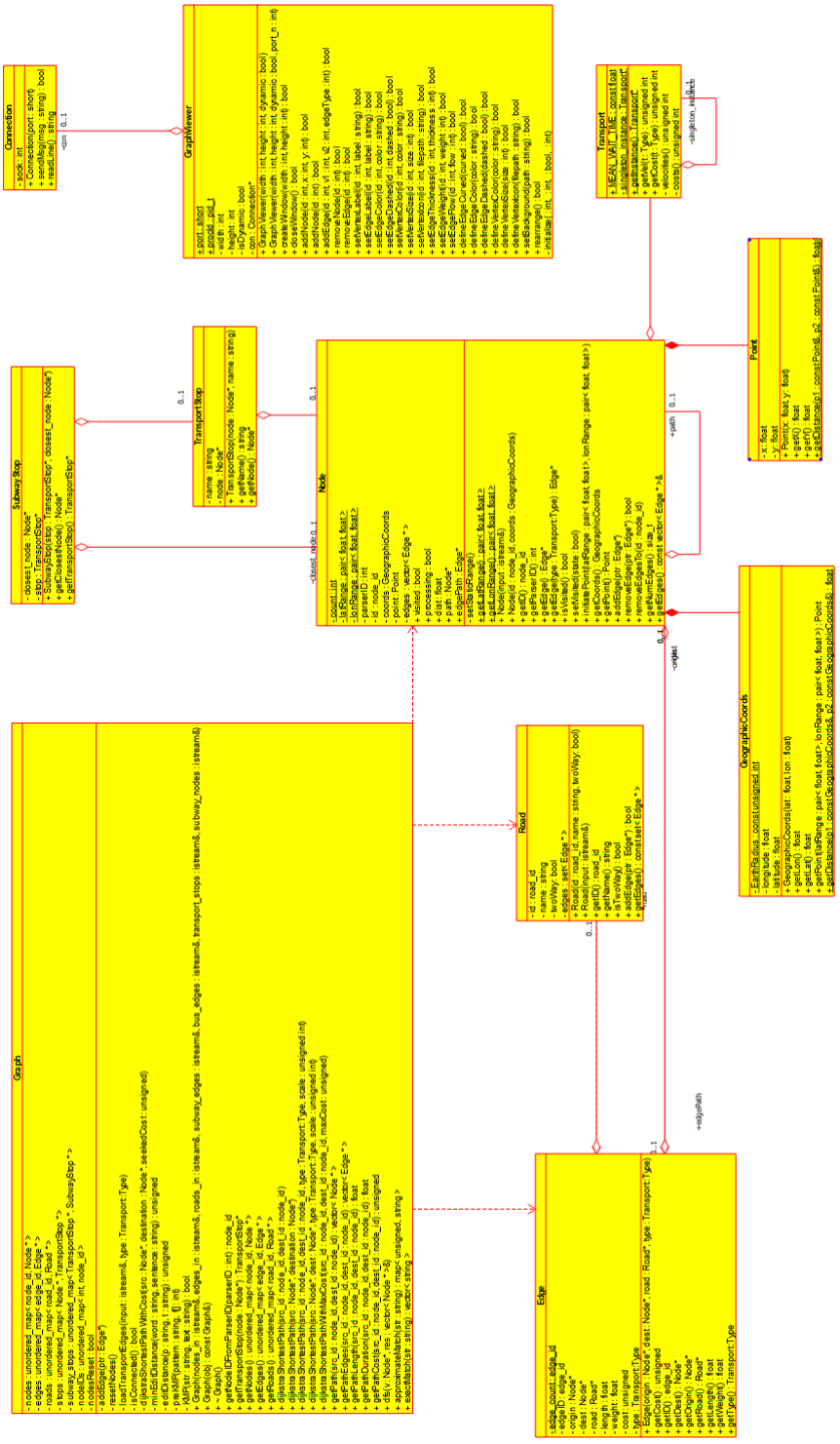


Figure 6: Class Diagram

6 Algorithm

6.1 Djikstra's Algorithm

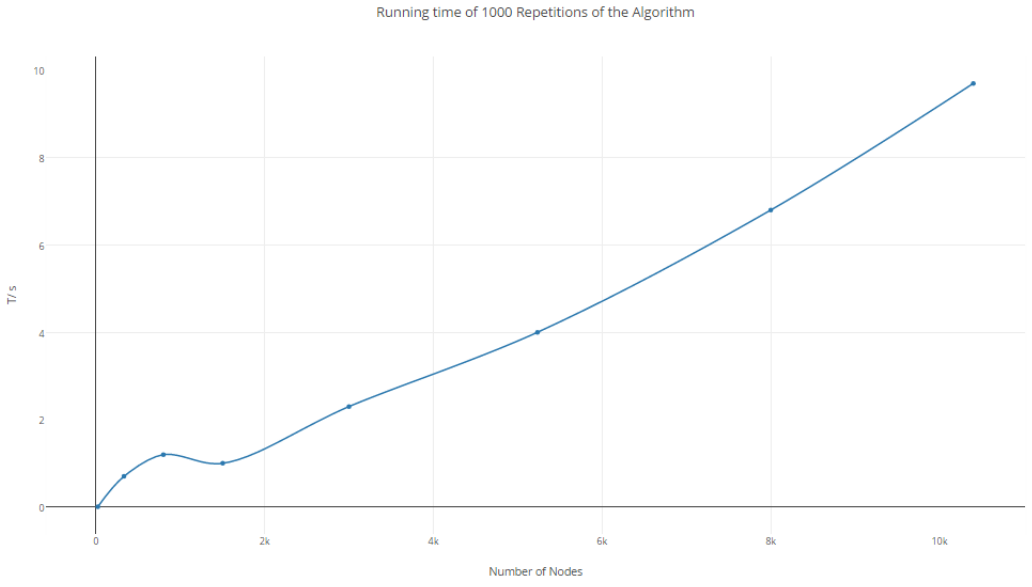
For the calculation of the optimal path we chose to use the **Djikstra's algorithm using a priority queue** however, since we have several case scenarios we had to adapt it to every single one. We opted for this algorithm since it is usually the most efficient one and has only one restriction which is that the values have to always be positive, which they are, so this limitation doesn't affect us. Dijkstra's algorithm keeps a set of all nodes yet to be processed. The size of this cluster is obviously linear to the number of nodes in the network, v. The algorithm does keep other local variables, but their size is constant with respect to both v and e.

In terms of complexity this algorithm is thus (where v is the number of nodes):

Temporal Complexity: $O(e * log(v))$

Spatial Complexity: $O(v)$

To verify that our system had this complexity we measured the execution time for several scenarios by changing the numbers of nodes and edges, ie, changing the graph size, and evaluating if the results are on par with what is expected from the algorithm.



Algorithm 1 Dijkstra's Algorithm Pseudo Code

```
1: function SHORTESTPATH(int src)
                ▷ Create a priority queue to store vertices that are being pre-processed.
2:     priority_queue < iPair, vector < iPair >, greater < iPair >> pq
3:     vector < int > dist(V, INF)   ▷ Create a vector for distances and initialize all
   distances as infinite (INF)
4:     pq.push(make_pair(0, src))          ▷ Insert source itself in priority queue and
   initialize its distance as 0.
5:     dist[src] = 0
6:     while !pq.empty() do                ▷ The first vertex in pair is the minimum
   distance vertex, extract it from priority queue. vertex label is stored in second of
   pair (it has to be done this way to keep the vertices sorted distance (distance must
   be first item in pair))
7:         int u = pq.top().second;
8:         pq.pop();                  ▷ 'i' is used to get all adjacent vertices of a vertex
9:         list < pair < int, int >>:: iterator i;
10:        for (i = adj[u].begin(); i != adj[u].end(); ++i) do   ▷ Get vertex label and
   weight of current adjacent of u.
11:            int v = (*i).first
12:            int weight = (*i).second
                                          ▷ If there is shorted path to v through u.
13:            if dist[v] > (dist[u] + weight) then          ▷ Updating distance of v
14:                dist[v] = dist[u] + weight
15:                pq.push(make_pair(dist[v], v))
16:            end if
17:        end for
18:    end while                          ▷ Print shortest distances stored in dist[]
19:    print("Vertex Distance from Source");
20:    for (int i = 0; i < V; ++i) do
21:        print(i, dist[i])
22:    end for
23: end function
```

6.2 Search Algorithms

In terms of implementing the search algorithm we used two different algorithms, one for the exact search and another one for the approximate search. For the exact search we use the Knuth-Morris-Pratt algorithm which makes use of previous match information that the straightforward algorithm does not by avoiding useless comparisons. While for the approximate search we use a proposed in classes algorithm that takes advantage of a dynamic programming matrix.

While the former has an overall complexity of: $O(|T|+|P|)$, due to it being a composed of two portions of the algorithm, the matcher and the compute prefix function, as shown below. (P - Pattern, T - Text)

While the latter has a complexity of: $O(P * T + P + T) = O(P * T)$

```
KMP-MATCHER(T, P)
 1  n ← length[T]
 2  m ← length[P]
 3  π ← COMPUTE-PREFIX-FUNCTION(P)
 4  q ← 0                                  ▷ Number of characters matched.
 5  for i ← 1 to n                         ▷ Scan the text from left to right.
 6       do while q > 0 and P[q + 1] ≠ T[i]
 7             do q ← π[q]                 ▷ Next character does not match.
 8          if P[q + 1] = T[i]
 9            then q ← q + 1               ▷ Next character matches.
10          if q = m                       ▷ Is all of P matched?
11            then print "Pattern occurs with shift" i − m
12                 q ← π[q]                 ▷ Look for the next match.
```

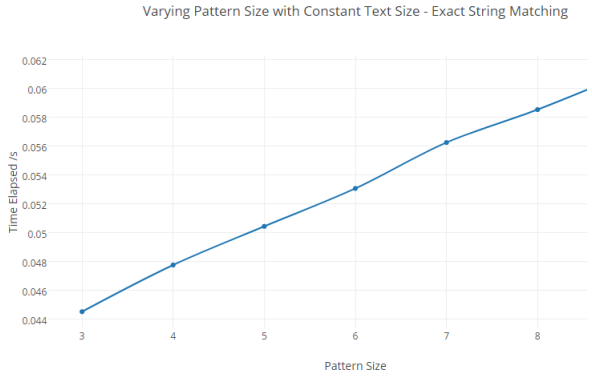
```
COMPUTE-PREFIX-FUNCTION(P)
 1  m ← length[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5       do while k > 0 and P[k + 1] ≠ P[q]
 6             do k ← π[k]
 7          if P[k + 1] = P[q]
 8            then k ← k + 1
 9          π[q] ← k
10  return π
```

Figure 7 Knuth-Morris-Pratt Algorithm's Pseudo Code

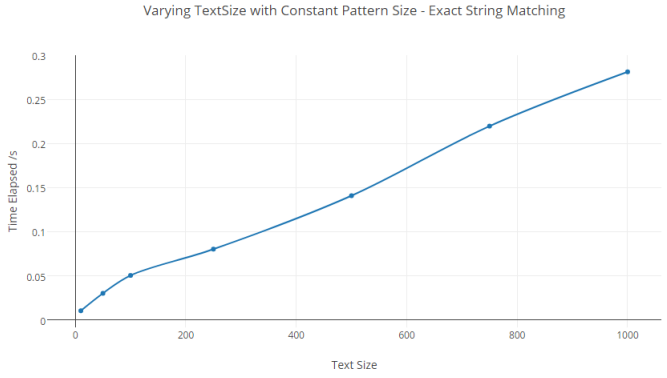
Algorithm 2 Approximate Search's Pseudo Code

```
1: function EditDistance(P, T)
                                                        ▷ Initialization
2:     i = 0 to —P— do D[i,0] = i
3:     for j = 0 to —T— do D[0,j] = j                   ▷ Recursive
4:     for i = 1 to —P— do
5:     for j = 1 to —T— do
6:     if P[i] == T[j] then
7:         D[i,j] = D[i-1,j-1]
8:     else D[i,j] = 1 + min(D[i-1,j-1],D[i-1,j],D[i,j-1])
9:     end if                                           ▷ Finalization
10:    return D‖P‖, ‖T‖
11: end function
```

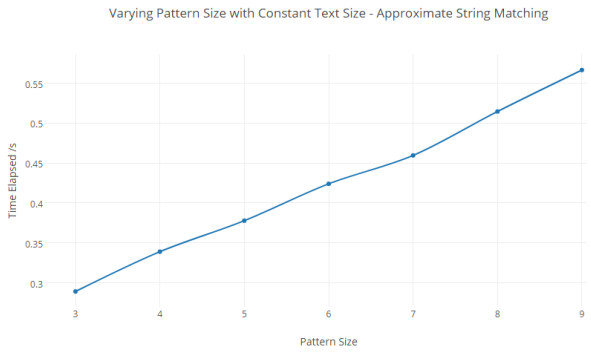
To be certain of this complexity in our implementation, we, again, measured the execution time for several scenarios by changing both the pattern size and the text size with both algorithms to verify if the results are on par with what is expected from the algorithm.



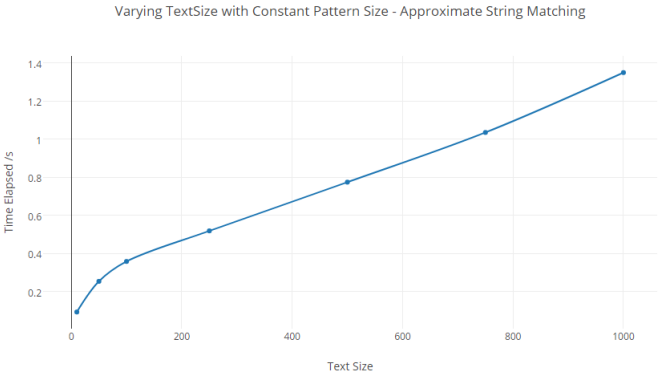
(a) Exact Search with Variable Pattern.



(b) Exact Search with Variable Text.



(a) Approximate Search with Variable Pattern.



(b) Approximate Search with Variable Text.

6.3 Connectivity

Since all the information we use as input comes from the exporting of the map information from *openstreetmaps* and extracting it through the parser, the graph should be connected, however we still verify if that is the case. For the verification of this connectivity we make the graph undirected, generating edges on the opposite side for each existing edge, and then we start a depth-first search. In addition we compare the number of nodes hit with the total number of nodes and if the they are equivalent it means the graph is fully connected. Otherwise, we expose the number and size of the clusters, sets of nodes, that make the graph not fully connected.

6.4 Results Analysis

Just as we predicted, the Dijsktra's algorithm has a logarithmic evolution through time. However, it is not fully perceivable just by the graph above since, for it to be easily visible we had to export an exorbitant number of nodes, but, as a result of the parser not extracting every tiny bit of information we want it would be an unnecessary task for us to manually go through. Despite this, we can still predict that the time the algorithm took to execute would increase, since Dijkstra's algorithm does not possess heuristics, and consequently, the more nodes it has, the more time it will take to process.

7 Implementation Difficulties

First and foremost, the main limitation to the work developed was the difficulty in obtaining real data from the OpenStreetMap parser, which leaves most of the information regarding transport stops unextracted, in particular the stop's name, location and line.
Another obstacle we came across was the conversion between geographical coordinates, provided by the openstreetmap files, and cartesian coordinates, to represent the Graph's data in a two dimensional plane. Eventually we decided on a simple approximate projection of the globe on a plane, as the area in question was far from the Earth's poles and, thus, could be performed with minimal error.
Regarding approximate pattern matching, an issue was raised of whether we should implement word by word matching or with the full search term. Ultimately, we decided to use both cases, regarding the pattern's edit distance as the minimum among word comparisons and sentence comparison.

8 Group Contribution

Overall, throughout the whole process of development of this project the workforce was evenly split between the three students, so that everyone could help towards the end goal, which we think was achieved with this project.

9 Conclusion

The aim of this report was to discuss a possible approach to the problem of planning a trip in an multi-modal transportation network, along with an optimal path connecting two points in such a network. A path is considered to be optimal if it meets a certain objective function. With the renowned Dijkstra's algorithm we were able to comply with the three objective functions successfully, some harder to implement than the others, but we still were able to implement them nonetheless. This task proposed to us in this course unit was definitely a good challenge that helped us understand the structure of graphs and its use cases along with search algorithms aimed at this very problems.
In conclusion, we consider that the intended objectives for this group project were met, either on an individual level or on a group level since now we have a more practical application of the contents taught to us in classes.

10 Bibliography

[1] http://www.openstreetmap.org/
[2] http://www.stackoverflow.com/