Faculdade de Engenharia da Universidade do Porto

TripMate: Multimodal Transportation Planner

Concepção e Análise de Algoritmos 2018/2019 - MIEIC:

Grupo A:

André Filipe Magalhães Rocha, up201706462@fe.up.pt

Carlos Jorge Direito Albuquerque, up201706735@fe.up.pt

João Miguel Ribeiro de Castro Silva Martins, up201707311@fe.up.pt

Index

1. Theme Description **4**

2. Problem Formalization **5**

2.1 Input Data 5

2.2 Output Data 5

2.3 Restrictions 6

2.4 Objective Function 6

3. Techniques and Algorithms to Implement **7**

3.1 Pre-processing 7

3.1.1 Approach 7

3.1.2 Pseudocode 7

3.2 Dijkstra's Algorithm 8

3.2.1 Pseudocode 8

3.2.2 Analysis 8

3.3 A* Algorithm 10

3.3.1 Pseudocode 10

3.3.2 Heuristics 11

3.4 Bidirectional search 12

3.4.1 General terminology and notation 12

3.4.2 Main approach 13

3.4.3 Pseudocode 13

3.4.4 Bidirectional search in Dijkstra and A* algorithms 14

3.5 Transit Node Routing 16

4. Supported Use Cases **17**

5. Implementation Plan **18**

6. Preliminary Conclusion **19**

7. Implementation **20**

7.1 Test Graph 20

7.2 Test Results & Empirical Analysis 21

7.2.1 Dijkstra vs. A* with heuristics 21

7.2.2 Unidirectional Vs. Bidirectional 22

7.2.3 Tests conclusion 24

7.3 Implemented Use Cases 24

7.3.1 Example 1 - From home to FEUP 25

7.3.2 Example 2 - Attending a class 25

7.3.3 Example 3 - Students also wander 26
7.3.4 Example 4 - Going further away 27
7.4 Graph Connectivity Analysis 28
7.4.1 Test Graph 28
7.4.2 Real Life Porto Map 28
8. Final Conclusion 30
9. Bibliography 31

1. Theme Description

TripMate is a multimodal transportation system with the objective of calculating the best way to reach all of the users destinations according to his/her schedule. It will be able to discover the fastest path between the user's origin point to all of his/her destinations during the day.

Adopting graph theory as our working basis, several algorithms will be implemented with the purpose of testing out their efficiency towards solving different issues, and determining which one is the most efficient in solving our current problem.

Since the purpose of this algorithm is to test its efficiency to solve real problems, the use of real maps information regarding transportation will be used in order to test out how it reacts to real world issues.

2. Problem Formalization

First things first, we must understand the problem we have in hands. The program will receive a **schedule** and make use of a **map** to advise the user on how to get to each appointment (with its own coordinates) **as fast as possible** by walking and by public transportation. It becomes obvious that we are working on a **point-to-point shortest path** problem - a well know problem in **graph theory**.

Now that we know what we are dealing with, it is time to formalize the data involved.

2.1 Input Data

- ➔ **B (V_B,E_B)**, **S (V_S,E_S)**, **F (V_F,E_F)** - three graphs, each with its own **vertex set (V)** and **edges (E)**;
- ➔ **B (V_B,E_B)** - graph representing the **bus** stops (vertices) and routes (edges);
- ➔ **S (V_S,E_S)** - graph representing the **subway** stops (vertices) and routes (edges);
- ➔ **F (V_F,E_F)** - graph representing road crossings (vertices) and streets (edges) to be traveled by **foot**;
- ➔ **V_i** - each vertex, in graph *i*, has:
    1. **coord** - pair with geographical coordinates (latitude and longitude);
    2. **visited** - boolean value indicating whether the vertex was already visited or not in a path search;
    3. **adj ⊆ E_i** - set of edges departing from V;
    4. **path** - previous vertex obtained from a path search
- ➔ **E_i** - each edge, in graph *i*, has:
    1. **source ∈ V_i** - vertex where the edge comes from;
    2. **dest ∈ V_i** - vertex where the edge ends;
    3. **weight** - time needed to travel the edge;
    4. **type** - mode of transport the edge refers to;
- ➔ **Appointment (A)** is composed by:
    1. **coord** - geographical coordinates of the appointment;
    2. **start** - starting hour of the appointment;
    3. **duration** - duration of the appointment;
    4. **path** - set of edges that represent the shortest path to reach the appointment. Initialized as *null*, relevant for output data;
- ➔ **Schedule (S_{in})** - user's daily schedule. Each schedule is made of:
    1. **start** - user's geographical coordinates at the start of the day;
    2. **end** - user's geographical coordinates at the end of the day;
    3. **appointments** - collection of appointments for the day.

2.2 Output Data

- ➔ **Schedule ($S_{oDataut}$)** - the user's daily schedule (S_{in}), but now ∀a ∈ appointments, a.path != null ∧ a.path is a double ended queue of edges representing a path from a coordinate to another.

2.3 Restrictions

- ➔ $\forall\ e \in E_i$ **: e.weight > 0** $\wedge$ **e.type = (‘subway’** $\veebar$ **‘bus’** $\veebar$ **‘foot’)** - the time must be a positive value and the type can only be one of the three values;
- ➔ $\forall\ a \in A$ **: a.duration > 0** - a duration of an appointment must be positive;
- ➔ $\forall\ a \in A, \exists\ v \in V_i$ **: a.coord = v.coord** - every appointment localization must exist in a map;
- ➔ $\forall\ s \in S_{in}, \exists\ v \in V_i$ **: s.start = v.coord, s.end = v.coord** - both the starting and ending location of a user must exist in a map;
- ➔ Let a_n be the *nth* appointment of a schedule S and a_{n+1} the next appointment in S. Knowing that $1 \leq n \leq$ (S.appointments.size - 1), $\forall\ a_n, a_{n+1} \in$ **S.appointments :** a_n**.start +** a_n**.duration <** a_{n+1}**.start** - there are no overlapped appointments;

2.4 Objective Function

- ➔ Our goal is to minimize the time spent in journeys between appointments. In other words, we want to minimize $\sum_{a \in S.appointments}$ **a.path.total_duration()**.

3. Techniques and Algorithms to Implement

TO CLARIFY:

A heuristic is a way to solve problems quicker and/or get an approximate solution when other methods are too slow or inaccurate. It trades optimality, accuracy and precision for speed.

A heuristic function ranks alternatives in search algorithms at each branch step based on available information to decide which branch to follow.

3.1 Pre-processing

In order to achieve better results with our algorithms we plan on pre-processing the given graphs in order to remove useless edges from it. To identify which edges will be useless in a specific case we shall be using an **heuristic** function in order to determine which edges can be dropped.

Our approach will be to determine which edges have a weight higher than a specific value, parameterized according to the distance (to be tested with several different values), which will consider edges that will most likely never be used for a given path, due to their big weight to cover that distance.

This approach has undoubtedly some flaws, since a perfectly fine edge to be taken in specific cases may be discarded; however, our goal will be to try and maximize this value, thus not losing any valuable edges and discarding a good amount of unviable edges at the same time.

3.1.1 Approach

Given our goal, we shall perform a **Depth-First Search (DFS)** in order to locate all edges and analyze their usability with an heuristic function relating their weight and distance.

3.1.2 Pseudocode

```
/* G - Graph, V - vertex, E - edges, Heu - Heuristic function */

performDFS(G, V):
    for Vertex v: V
        visited(v) <- false

    for Vertex v: V
        if not visited(v):
            visitNodeDFS(G, V)


visitNodeDFS(G, V):
    visited(v) <- true
    for Edge w: Adj(v)
        if(Heu(w))
            removeEdge(w)
        else if not visited(w.dest)
            visitNodeDFS(G, w.dest)
```

3.2 Dijkstra’s Algorithm

The classic algorithm for finding the shortest path between vertices in a graph is Dijkstra's algorithm. Named by its creator, it was firstly published in 1959. The variant being used for our purpose is the original one, to find the shortest path between two vertices. However, along the way the algorithm also finds the shortest path for each vertex until the destination, which makes him a single-source shortest path - finding every node’s distance to the source until we find the destination’s distance - algorithm as well.

3.2.1 Pseudocode

```
/* G - Graph, V - vertex, E - edges, Vi - initial vertex, Vd - destination vertex*/

Dijkstra(G, Vi, Vd):
    for Vertex v: V
        distance(v) <- INF
        path(v) <- nil
    dist(Vi) <- 0
    Q <- {} // min-priority queue
    Insert(Q, (Vd, dist(Vi)))
    while Q != {}
        v <- Extract-Min(Q) // Minimum being the one with the lowest score

        if equals(v, Vd):
            return;

        for Edge w: Adj(v)
            if dist(w) > dist(v) + weight(v, w) then
                dist(w) <- dist(v) + weight(v, w)
                path(w) <- v
                if w not in Q then
                    Insert(Q, (w, dist(w)))
                else
                    Decrease-Key(Q, (w, dist(w)))
```

3.2.2 Analysis

The algorithm can be divided in three big parts:

1. **Graph initialization** - every vertex has an infinite distance to the source and no path (vertex from where you reach the current one), this way the first time we get to a vertex both the distance and path will be updated accordingly;
2. **Structure initialization** - the starting vertex gets a distance of zero and is inserted in a ordered data structure (in the snippet of code above a minimum priority queue, but heaps or Fibonacci heaps are also commonly used) that will collect all vertex already found by a path and keep the one with the lowest distance ready to be processed next;
3. **Processing / Main Cycle** - until we reach our destination or we run out of vertices in the graph, the algorithm checks for each neighbour (N) of the next vertex (V) in the queue if its distance

becomes shorter by going from V to N; if so, the distance and path of N are updated and the data structure is rearranged, otherwise there is no change needed.

Notice that Dijkstra's algorithm will go through a huge amount of vertices because, as we move away from the source, distances get bigger and the algorithm has to go back to a previous vertex to check if there really is no better way to get the other vertices.

Thereby, Dijkstra's algorithm has the ability to always present us with the most optimal solution. However, it still is a huge cost that can make it quite slower than more expedient algorithms, such as A* presented in the next page.

Dijkstra's algorithm correctness is proved by **induction on the distance of vertices**[1]. Its **temporal complexity** is **O((|V| + |E|) * log|V|)**: O(|V| * log|V|) that results from operations in the priority queue, which are made |V| times and each made in logarithmic time in |V| (size of the queue), summed with O(|E| * log|V|) that comes from the decrease-key operation, which is done a maximum of |E| times and takes log|V| each time it is performed. On the other hand, the **spatial complexity** is **O(|V|)** that is the maximum size of the queue used.

[1] Glencora Borradaile, *Dijkstra's Algorithm: Correctness by induction, 2005*

3.3 A* Algorithm

It is intended to study the use of the A* star algorithm as a means of comparison against the Dijkstra algorithm. Firstly introduced by **Peter Hart, Nils Nilsson and Bertram Raphael** during their time at Stanford Research Institute, first published on the year of 1968. This algorithm is considered to be an extension of Dijkstra's, due to the fact that it uses an heuristic to guide its search towards the destination vertex.

3.3.1 Pseudocode

```
/* G - Graph, V - vertex, E - edges,
Vi - initial vertex, Vd - destination vertex, Heu - Heuristic function */

A_star(G, Vi, Vd, Heu):
    for Vertex v: V
        distance(v) <- INF
        path(v) <- nil
        score(v) <- INF
    dist(Vi) <- 0
    score(Vi) <- Heu(Vi, Vd)
    Q <- {} // min-priority queue
    Insert(Q, (Vd, dist(Vi)))
    while Q != {}
        v <- Extract-Min(Q) // Mininum being the one with the lowest score

        if equals(v, Vd):
            return;

        for Edge w: Adj(v)
            if dist(w) > dist(v) + weight(v, w) then
                dist(w) <- dist(v) + weight(v, w)
                score(w) <- dist(w) + Heu(v, Vd)
                path(w) <- v
                if w not in Q then
                    Insert(Q, (w, dist(w)))
                else
                    Decrease-Key(Q, (w, dist(w)))
```

As it can be seen, this algorithm is very **similar to the Dijkstra's,** with the slight difference that it uses an **additional heuristic** to search for the best possible vertex at a given point.

3.3.2 Heuristics

Due to nature of our search, being for the shortest path between two geographical points, three heuristics shall be considered:

- Euclidean Distance:

```
/* p1 - point, p2 - point */
euclidean(p1, p2):
    return sqrt(square(p1.x - p2.x) + square(p1.y - p2.y))
```

- Chebyshev Distance:

```
/* p1 - point, p2 - point */
chebyshev(p1, p2):
    return max(abs(p1.x - p2.x), abs(p1.y - p2.y))
```

- Manhattan Distance:

```
/* p1 - point, p2 - point */
manhattan(p1, p2):
    return (abs(p1.x - p2.x) + abs(p1.y - p2.y))
```

It is known that the A* **may not guarantee an optimal solution** (contrary to Dijkstra's algorithm), due to the heuristic function being used. Thereby, we are going to test **several heuristics** related to distance calculation algorithms. With that, we are going to try to identify which of those has a better performance on our problem.

If the heuristic function is **admissible**, i.e., it never **overestimates the effort to reach to the goal** then it is **non-decreasing**, thus guaranteeing the optimality of the solution.

In terms of complexity, it is known that the this algorithm is identical to Dijkstra's in every way, except for the calculation of the weights, therefore its complexity is the same. Hence, its **temporal complexity** is **O((|V| + |E|) * log|V|)** and its **spatial complexity** is **O(|V|)**.

3.4 Bidirectional search

Bidirectional search is a **graph search algorithm** dedicated to solve the **shortest path** problem in a **directed graph**. Basically, it performs **two searches at the same time**: one regular search from the starting node, and one inverse, starting from the end node.

The search is complete when the searches meet. The big advantage of this approach is its rapidity. In a generic search problem complexity, where both searches expand a tree with, say, a branching factor **b**, and the real distance from start to destiny is **d**, each search has complexity $\mathbf{O(b^{d/2})}$, thus making the total complexity for this problem the sum of both. This result is quite lower, i.e. better, than the complexity resulting from a single search from the beginning, which would have complexity of $\boldsymbol{O(b^d)}$**.**

This algorithm is believed to improve the performance on the already optimal search algorithms (in most cases), **Dijkstra's** and **A***.

3.4.1 General terminology and notation

b the branching factor of a search tree

k(n,m) the cost associated with moving from node *n* to node *m*

g(n) the cost from the root to the node *n*

h(n) the heuristic estimate of the distance between the node *n* and the goal

s the start state

t the goal state

d the current search direction. By convention, *d* is equal to 1 for the forward direction and 2 for the backward direction (Kwa 1989)

d' the opposite search direction (i.e. *d' = 3 - d*)

$TREE_d$ the search tree in direction *d*. If *d = 1*, the root is *s*, if *d = 2*, the root is *t*

$OPEN_d$ the leaves of $TREE_d$. It is from this set that a node is chosen for expansion.

$CLOSED_d$ the non-leaf nodes of $TREE_d$. This set contains the nodes already visited by the search

3.4.2 Main approach

The main approach for Bidirectional Heuristic search used generically is the **Front-to-Back** (or Front-to-End).

It consists of calculating the *h* value of a node *n* by using the heuristic estimate between *n* and the root of the opposite search tree, either *s* or *t*.

Another known approach is the **Front-to-Front**, where the *h* value of a node *n* is calculated using the heuristic estimate between *n* and a subset of $OPEN_d$. However, this approach is rather computationally demanding because every time a node *n* is put into the open list its $f = g + h$ value must be calculated, which involves calculating a heuristic estimate from *n* to every node in the opposing *OPEN* set, as described above. The *OPEN* sets increase in size exponentially for all domains with $b>1$.

3.4.3 Pseudocode

```
/* s - source node, t - destination node,V - graph vertices */
bidirectionalSearch(s, t) :

    Vertex intersectNode <- null

    // necessary initialization
    for Vertex v: V
        visited(v) <- false
        path(v) <- null

    Q1 <- {} // initialize the forward search queue
    Insert(Q1, s)
    visited(s) <- true

    Q2 <- {} // initialize the backward queue
    Insert(Q2,t)
    visited(t) <- true

    while Q1 != {} && Q2 != {}

        // do desired search from source and destination vertices
        forwardS(Q1)
        backwardS(Q2)

        // check if there is a node that have
        // been visited in both searches
        intersectNode <- isIntersecting();

        // if intersecting vertex is found
        // that means there is a path
        if(intersectNode != null)
            return SUCCESS

    return ERROR
```

3.4.4 Bidirectional search in Dijkstra and A* algorithms

The **Dijkstra** and **A*** algorithms represent two of the best search algorithms currently being used. Their **exactitude** and **speed** in relation to other algorithms makes them the best for finding **optimal solutions**. In order to **improve this performance**, the **bidirectional search technique** has been applied to these algorithms and the found results have been approving of this theory.

In a paper published in the ACM SIGSPATIAL Cup 2015 (that achieved the 3rd place), regarding path search algorithms with restricted areas, these algorithms are exploited and tested.

The algorithms on focus were the Dijkstra, A*, bidirectional Dijkstra and bidirectional A*.

The **bidirectional Dijkstra** algorithm uses the **Dijkstra** algorithm on **both ends of the path**, i.e. the **source** and **destination** vertices. Search from the source is viewed as forward and from the destination, backward. While the forward source is the traditional Dijkstra, the backward search uses the inverse edges. $E_i = ((i,j)|(j,i) \in E)$.

Both searches aim to meet in middle of the source and destination. The **final cost** of the path is read as the **sum of the distances of both ends** to the meeting point. That is, $dist_f(q) + dist_b(q)$, where q is the meeting point, f the forward distance and b the backward distance. In case there are multiple meeting points, where the one which presents the minimal sum of said distances is chosen.

Since the A* is in itself a variation of Dijkstra's, the **Bidirectional A*** follows the same logic that Bidirectional Dijkstra. The **goal** is to get the **A* cost** for this technique.

The tests consisted of computing 100 shortest path searches, evaluating two different cases.

The first uses the edge length as a value for edge cost (meaning real shortest path) and the second uses the edge travel time as the edge cost (meaning fastest path search).

The tests performed showed the following results:

Method	execution time [μs]	maximum error [%]	average error [%]
Dijkstra	8747	0.0	0.0
Bi-Dijkstra	7327	0.0	0.0
A*	3478	25.3	9.4
Bi-A*-Dij	3462	24.3	8.3
Bi-A*	2797	28.3	10.1

Table 1: Result of the shortest path search

Method	execution time [μs]	maximum error [%]	average error [%]
Dijkstra	8961	0.0	0.0
Bi-Dijkstra	6800	0.0	0.0
A*	2995	40.0	19.8
Bi-A*-Dij	3017	39.4	18.5
Bi-A*	2386	40.0	22.2

Table 2: Result of the fastest path search

In both tests the execution time is measured (in microseconds) and the maximum and average errors are taken into account (in percentage).

The **conclusion** that fruited from this experience was that **best/fastest exact method** was the **Bidirectional A*** while the **slowest** was **Dijkstra**.

Further details are explained in the paper[2].

[2] František Kolovský and Jan Ježek, *Restricted areas obeying path search algorithm without preprocessing*, 2015

3.5 Transit Node Routing

The **transit node routing** approach on shortest path search has a rather **useful applicability in the context of our problem**.

This approach can be first understood in a intuitive observation used by people when travelling from a place to another. When the t**wo locations are relatively afar** from one another, we tend to travel using a **small subset of the available routes** that are relevant to long-distance trips (e.g.: **highways**).

So, when calculating the shortest path between two points, we need first to **preprocess** some connections: between the **source** or **destination** and its **access** transit nodes and between **all pairs of transit nodes**. This is because:

1. when going from the **source** (or **near** it) to a **destination** (or vice versa) that is considered **far**, it will **always use** one of the **access points** of the **preprocessed set**;
2. for that set of **points near the source** (i**ncluding the source itself**), the **same set of access points** is **shared** among them, which means that we are able to **save resources** in an already **heavy task** like **pre-processing**.

This technique must be taken into account because while it does **bring advantages** calculating the best path, as it was mentioned before, it **bears a quite heavy burden** in **memory consumption**.

Now, bringing this technique to the context of our problem, it is also intuitive to relate the **pedestrian routes to local routes**, and whenever we **reach a bus stop or a subway station** we **access faster routes**, being those the **bus and subway routes**, which means that the **bus stops and subway stations represent our “access points” or “transit nodes”**. It’s only a matter of scaling, since walking from end-to-end in a city and doing the same using buses or subways are rather similar to highways and local roads, being this method strongly connected to the highway network hierarchy analogy.

4. Supported Use Cases

This application will be able to provide information given an **user's schedule**, with all the information needed in order to calculate said solution (a location of the activity, respective beginning time and duration must be provided for each activity). Given this information, considering that all activity's locations do exist in the provided map, it will be calculated the fastest way to reach every destination provided, giving the necessary information for that purpose.

Naturally, more than one map will be provided, being that some may be fake (as in produced by us for testing), others may happen to be actual real maps.

Moreover, since there's a support for the calculation of a user's whole schedule, there will also exist the possibility to ask for unique directions from one point to another, without the need to prepare a whole schedule of trips.

In addition, the application will present the user with a map and the path routes highlighted to help the user understand every path he has to take.

In a later stage of the project, we intend to add the possibility to swap between minimizing the time or the money spent for the trip. We also plan to take into account available bus and subway schedules to make a more robust and helpful application for daily use.

5. Implementation Plan

After this first part where we researched the best ways to tackle our problem, the following part will be the implementation of our solution. But first, we arranged a plan for what our priorities should be and how we are going to proceed with each step, sequentially.

1. Structure of graph manipulation
2. Apply the algorithms in simple problems (with testing and analysis)
3. Built structure of graph in the problem's context
4. Implement Graph Viewer to visualize the solution in a map
5. Optimize the solution and register results for comparison and analysis
6. Test a real-life map and route planning in the problem's context
7. Introduce trip expense to the problem variables (optional)
8. Introduce public transportation schedule to the problem variables (optional)
9. Introduce transportation services delay to the problem variables (optional)

The topics **1 to 6** we intend to fully explore while the rest (**7 to 9**) are functionalities that have been thought over and shall be explored within the time we have left in a later stage of our project development.

6. Preliminary Conclusion

With this research our main goal is to define which tools we will use to implement TripMate. Thereby, we hope that it is clear why we focus certain algorithms while nor even mentioning others.

While not deeply analyzed, all used algorithms were fundamented and explained. It was not our intention to go through an extensive and detailed description of each one, but to give the reader a nice summary of how things work and how they will work to solve our problem.

Moreover, such analysis presented us with the ability of discerning which algorithms we expect to have a better performance for our problem, given that we expect to have significant optimization, while using the A* algorithm over Dijkstra's, as well its Bidirectional approach over unidirectional one.

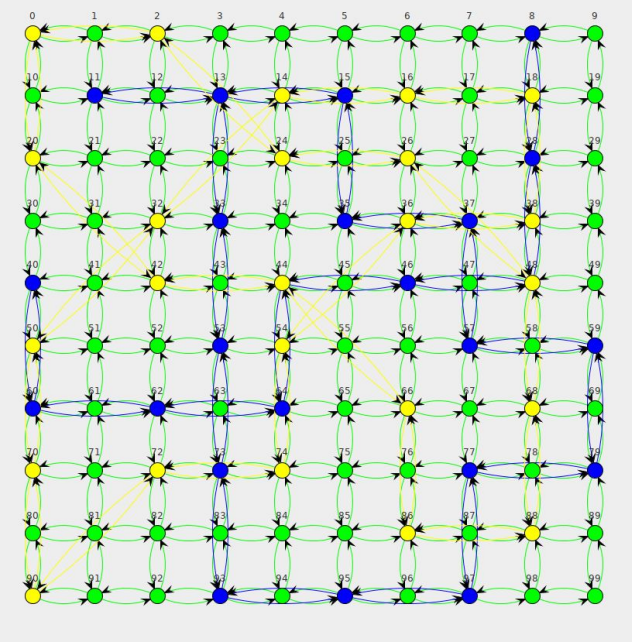
Finally, all the research and foundation contributed in great measure for our understanding of the subject and for a very good preparation to the actual implementation of our application.

7. Implementation

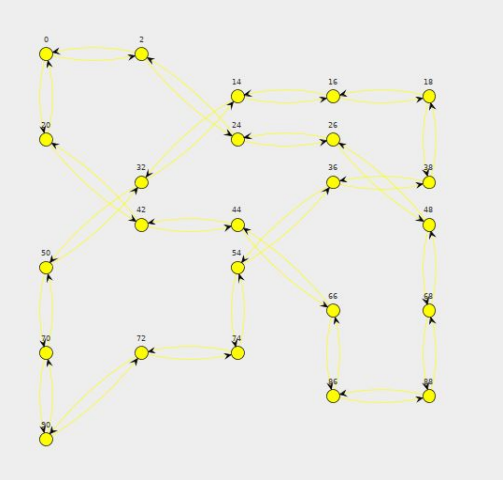
After this first part was concluded, we know entered the more practical aspect of our project. The main ideas were already set, the algorithms chosen, so we started to code our solution. Since none of the algorithms were invented by any of us, it came rather easy to have something to work with. Initially we adapted the Graph solution given in the practical classes to better suit our needs, and then apply the desired algorithms and not very much late so we started to get results. It was time to implement a test case and do an analysis on these empirical data.

7.1 Test Graph

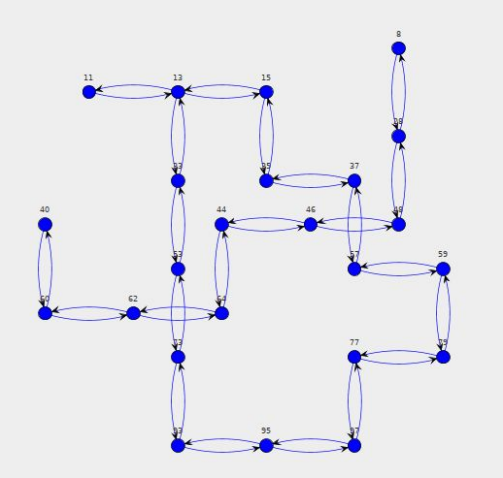
To test out our solution, we implemented a test graph adapted from one given in classes, with the addition and distinguishment on the graph edges. Now we had edges for pedestrian routes (colored green), bus routes (colored blue) and subway routes (colored yellow). The test graph was built as a grid NxN, N being a number that is provided in the test as the grid side length. To test how the solution handled escalation, we used various N. Below shows an general overview of the test graph structure, on a 10x10 grid. It is shown as well the transport routes simulated by the edges.



Test graph example:: 10x10 grid with the different transport options highlighted accordingly



Subway routes: two routes highlighted in yellow



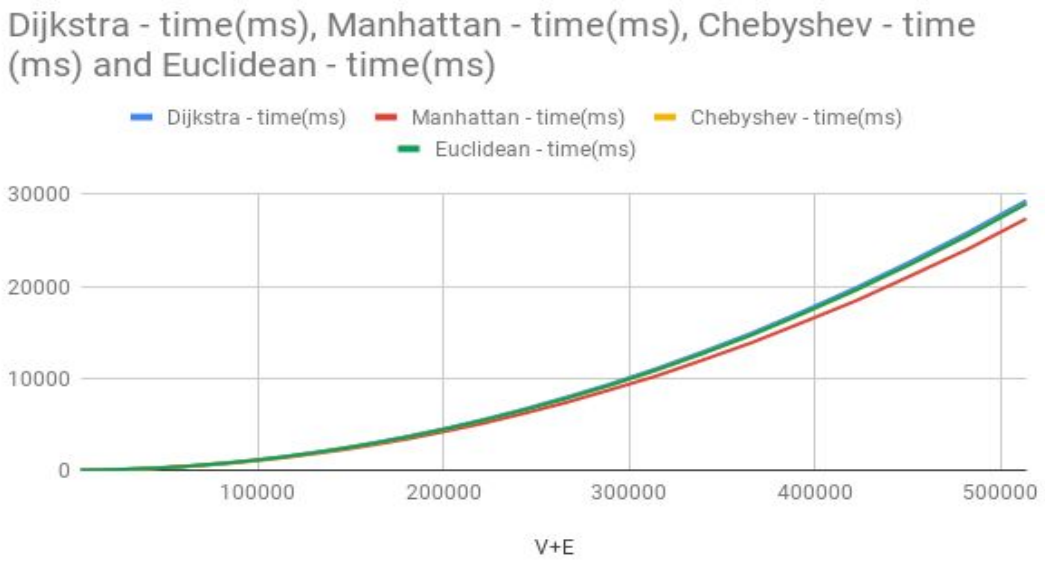
Bus routes: three routes highlighted in blue

7.2 Test Results & Empirical Analysis

After having our test case built we started to put our algorithms to good use and run them in different scenarios and collect data on their temporal performance.

7.2.1 Dijkstra vs. A* with heuristics

Given the similarity expressed earlier in this report between these two algorithms it was possible for us empirically compare Dijkstra's algorithm with A* and the 3 heuristics presented above. It is important to notice that the tool developed by us, inspired by one provided by the teachers, creates **grid graphs**, thus influencing some of the the heuristics in detriment of others. This information is more realistically compared to cities with a grid plan, such as Lisbon or Manhattan. The information captured is detailed below:



V+E	Dijkstra - time(ms)	Manhattan - time(ms)	Chebyshev - time(ms)	Euclidean - time(ms)
4654	3	2	3	3
8204	9	8	8	9
12754	20	19	19	20
18304	41	38	39	41
24854	74	76	88	73
32404	125	120	121	132
40954	205	181	203	194
50504	305	278	295	294
61054	437	400	421	425
72604	629	566	599	597
85154	838	767	812	819
98704	1116	1031	1087	1093
113254	1461	1348	1428	1436
128804	1889	1753	1846	1857
145354	2391	2211	2343	2351
162904	2999	2786	2943	2952
181454	3718	3436	3640	3650
201004	4551	4233	4470	4480
221554	5518	5113	5425	5434
243104	6638	6168	6526	6538
265654	7921	7342	7786	7797
289204	9354	8721	9215	9222
313754	11000	10196	10837	10842
339304	12864	11965	12686	12687
365854	14917	13872	14715	14719
393404	17268	16083	17027	17051
421954	19845	18426	19583	19602
451504	22704	21149	22425	22428
482054	25835	24025	25512	25512
513604	29315	27348	29008	28976

Based on these results, it is possible to conclude how the A* does perform much better than Dijkstra's, whenever the heuristics is fitting for the problem. Based on this information, it is possible to conclude that Chebyshev Distance is a poor heuristics for this algorithms, being that Manhattan had the best performance, once again, deduced by the fact that the graph was a perfect grid.

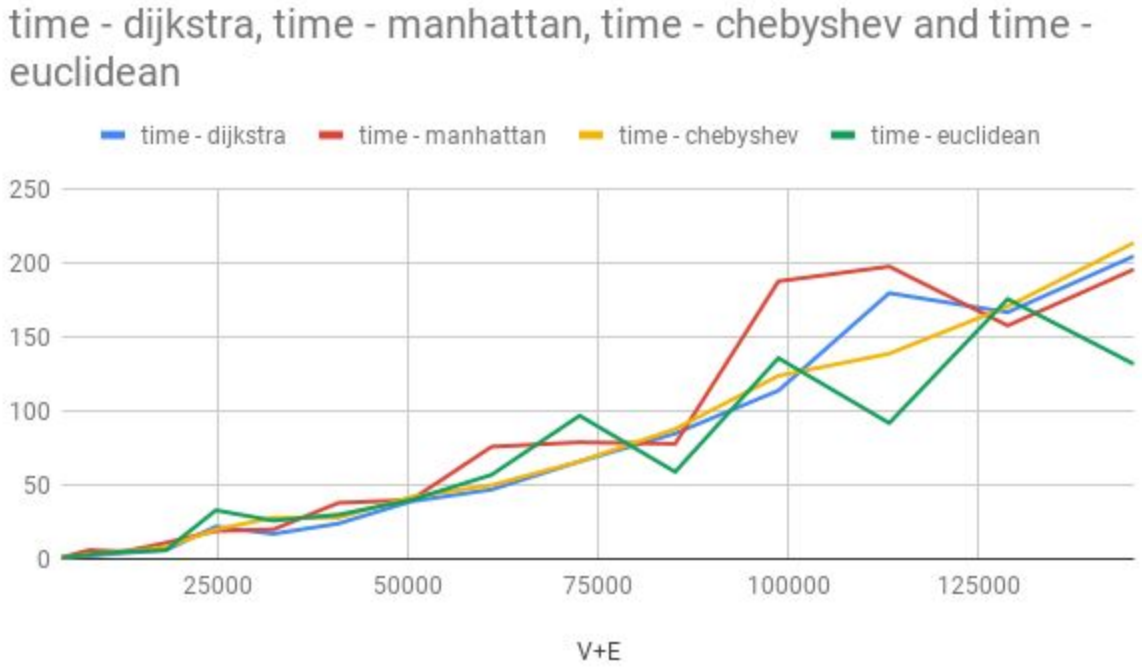
The performance was the expected in this case due to the nature of the graph, euclidean distance would not perform as properly as Manhattan, even though there was a slight optimization.

7.2.2 Unidirectional Vs. Bidirectional

Our goal was to empirically compare the results of the algorithms displayed above with their bidirectional implementation.

As expected, results show that it is much faster to use any of the bidirectional approaches to algorithms than their normal implementations.

It is relevant to firstly explain how our Bidirectional algorithm resorts to a **multithreading approach**, in order to parallelize work even further, thus making that our solution does have quite a **significant gain** against their unidirectional approach, partially due to this parallelization.



V+E	time - dijkstra	time - manhattan	time - chebyshev	time - euclidean
4654	1	1	1	1
8204	2	6	3	3
12754	4	5	5	5
18304	6	11	8	6
24854	22	19	20	33
32404	17	20	28	26
40954	24	38	28	30
50504	39	40	42	40
61054	47	76	50	57
72604	66	79	66	97
85154	85	78	88	59
98704	114	188	124	136
113254	180	198	139	92
128804	167	158	171	176
145354	205	196	214	132

It is important to notice that the graphs used for these tests are exactly **the same** as in the unidirectional data. Hence, data is comparable against their unidirectional approaches, since they were looking exactly for the same result.

Unexpectedly, the performance of the manhattan heuristics was much more unstable than its Unidirectional approach, euclidean being the one to show the better results. Opposing the unidirectional algorithms, bidirectional algorithms seem to have a much more unstable increase, even if those results are much better than their original approaches, being that Dijkstra's algorithm is sometimes even better than A*'s.

7.2.3 Tests conclusion

As expected, after implementing all of the algorithms described above, it was possible to obtain results similar to the expected: the A* algorithm is significantly better than Dijkstra's, especially when paired with a **good non-decreasing heuristics**. It was possible to obtain that Chebyshev's distance heuristics is not as appropriate for the job as expected, being that Manhattan's distance and Euclidean distance do add significant improvement towards the algorithm, being the more fitting to solve shortest path problems.

Furthermore, it was possible to observe significant improvement with the use of Bidirectional approach to the issue. Even if it is accounted the fact that it does take some time to invert the graph, as well as check if a point in common has been found, this is all mostly insignificant when compared to the gain.

7.3 Implemented Use Cases

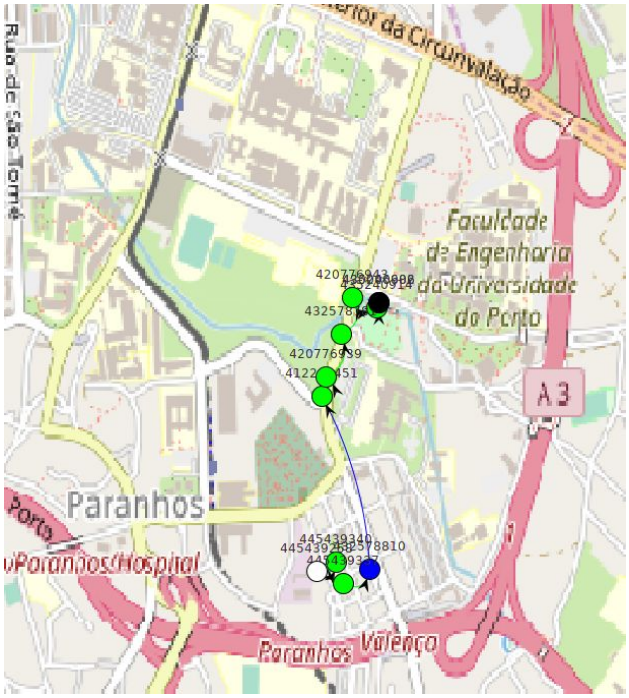
Once our algorithms were put up to testing and empirical results were taken, it came the time to implement our solution in the project's theme context. Given several factors, including time available and information provided, we decided to set up our demonstration in the scenery of the city of Porto, where the faculty is located and where we currently study.

In order to provide a general view of what our solution has to offer we have selected four examples of user schedules and appointments and the routes generated displayed with GraphViewer, with our graph shown on top of imagery of the city's schematics.

To help identify the main points of the routes we have identified in the graph the origin, destination and middle points with white, black and red colored vertices respectively. The examples are shown below:

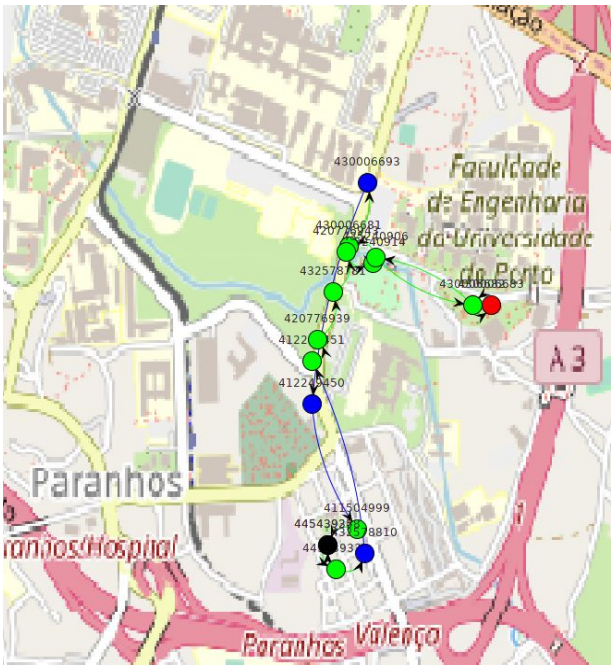
7.3.1 Example 1 - From home to FEUP

The usual path of a FEUP student - a simple point to point route.



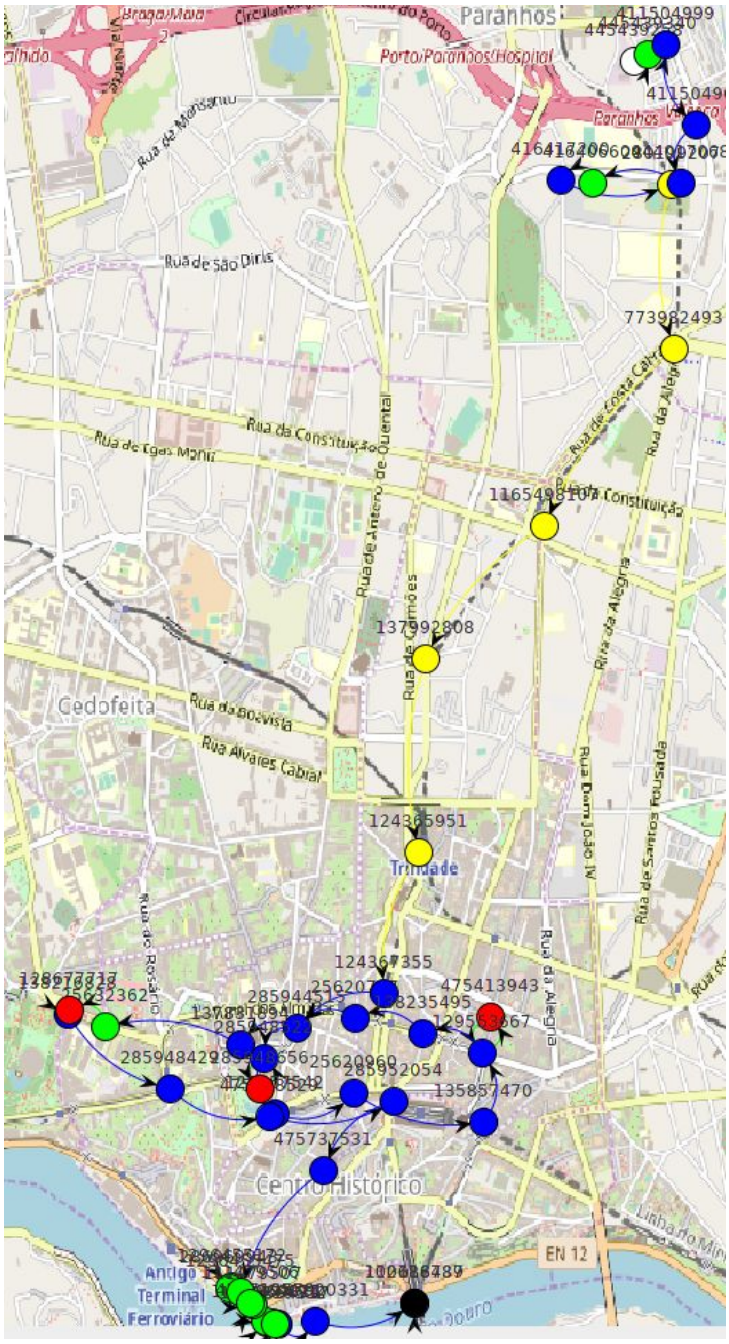
7.3.2 Example 2 - Attending a class

Actually, most of the times a student goes to the faculty to attend a class (an appointment, marked red on the map). Hence, the student goes from home to FEUP to attend the CAL class and goes back home after it is finished.



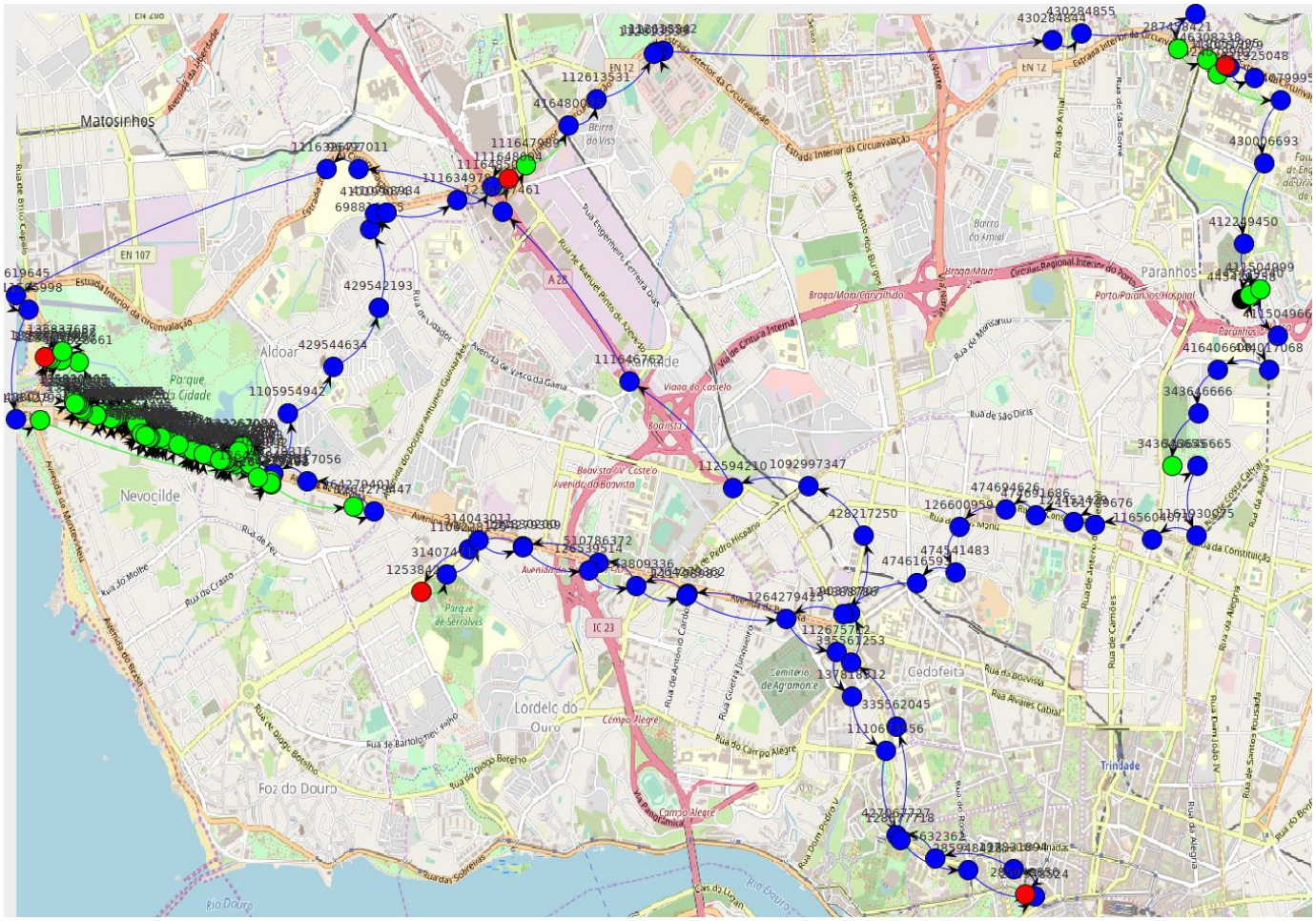
7.3.3 Example 3 - Students also wander

Now, going from home to Porto's downtown, the student has to get the subway and can happily take a walk in Palacio de Cristal gardens. Of course this time the student ends the day watching the sunset at bridge D.Luis.



7.3.4 Example 4 - Going further away

Last but not least, the student decides to visit Serralves, have lunch downtown, go for a walk by the sea, dine at Norteshopping and get a drink with his friends near São João before going home exhausted.

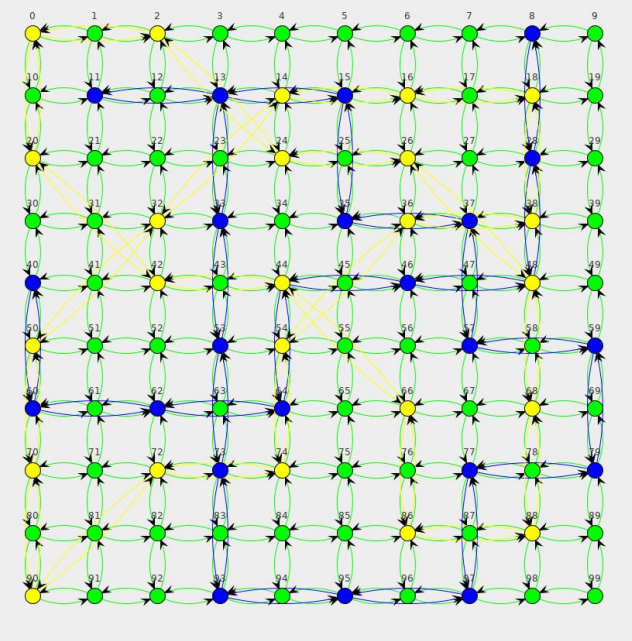


7.4 Graph Connectivity Analysis

Due to the nature of our implementation, it is important to make two sets of analysis of the Graph Connectivity: one for the graph tool developed for empirical testing, and for the real case data gathered by OpenStreetMaps and the STCP Bus API.

7.4.1 Test Graph

In order to make a good analysis lets take a look back at our test graphs:



As it can be seen, this graph has a very high density of edges, being able to connect every single vertex to any other vertex in the graph, thus indicating it is a strongly connected graph. This information also reveals why such heuristics as Manhattan's distance do have a **strong impact** in the performance of the search.

7.4.2 Real Life Porto Map

Since this graph is based on real life information of Porto's topography, it was needed to use search algorithms to determine better its connectivity. For this, it was used a **DFS algorithm** (Depth-First search) which was able to determine how the graph behaved when trying to reach all Vertices from a specific one chosen randomly.

```
Num of vertex's in graph: 10176
Num of vertex's found through dfs: 8543
```

As shown in the image above, a great deal of vertices aren't found through Depth-First search, indicating that the Graph is **not connected**. Due to the high amount of vertices not found, it is also possible to indicate that is due to the fact that the information provided by OpenStreetMaps is not complete; however it is important for us to understand this lack of information, and how to improve our algorithms from it.

8. Final Conclusion

Reaching the end of our project development we reflect on the topics thought over in an earlier stage and the goals that we set out to be satisfied by ourselves. Indeed we did verify that most of these goals were met, since all those who were considered fundamental to our project's structure were completed successfully.

Besides that, it is also worthy to refer our testing cases, as we used highly complex graphs to test our algorithms and draw more valuable data from it making a stronger, empirical, conclusion.

At last, even though we ended up not having a more interactive interface as the theme's description suggested we should have, what we produced for our demo does meet the description requirements, as it simulates the usage of the application in real life scenarios.

To wrap it up, we gladly admit to be satisfied with the results we obtained and the final product of our work. Every member contributed with great effort to the state of our application and the robustness of our algorithms and approach. Thanks to a strong communication and mutual aid, each member contributed equally for this project.

9. Bibliography

- A* search algorithm, https://en.wikipedia.org/wiki/A*_search_algorithm.
- Auer, Andreas and Hermann Kaindl. *A Case Study of Revisiting Best-First vs. Depth-First Search*.
- Bast, Holger and Stefan Funke and Peter Sanders and Dominik Schultes. “Fast routing in road networks with transit nodes.” *Science* 316, no.5824(2007): 566. doi:10.1126/science.1137521.
- Bidirectional Search, https://en.wikipedia.org/wiki/Bidirectional_search.
- Borradaile, Glencora. *Dijkstra's Algorithm : Correctness by induction*. Oregon State University. Available at: https://web.engr.oregonstate.edu/~glencora/wiki/uploads/dijkstra-proof.pdf.
- Champeaux, Dennis. "Bidirectional heuristic search again". *Journal of the ACM* 30, no. 1(1983): 22–32. doi:10.1145/322358.322360.
- Champeaux, Dennis and Sint, Lenie. "An improved bidirectional heuristic search algorithm". *Journal of the ACM* 24, no. 2(1977): 177–191. doi:10.1145/322003.322004.
- Goldberg, Andrew V. and Chris Harrelson and Haim Kaplan and Renato F. Werneck. *Efficient Point-to-Point Shortest Path Algorithm*. Princeton University. Available at: https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf.
- Javaid, Adeel. (2013). *Understanding Dijkstra Algorithm*. SSRN Electronic Journal. Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2340905.
- Kolovský, F. and J. Ježek. *Restricted areas obeying path search algorithm without preprocessing*. Kolovsky.cz. Available at: https://kolovsky.cz/static/Restricted_areas_obeying_path_search_algorithm_without_preprocessing_with_copy.pdf.
- Kumar, Atul. “Bidirectional Search.” Geeks for Geeks. https://www.geeksforgeeks.org/bidirectional-search/.
- Lyfat. “Euclidean vs Chebyshev vs Manhattan Distance.” Lyfat. May 22, 2012. https://lyfat.wordpress.com/2012/05/22/euclidean-vs-chebyshev-vs-manhattan-distance/.
- Ortega-Arranz, Hector and Diego R. Llanos and Arturo Gonzalez-Escribano. *The Shortest-Path Problem: Analysis and Comparison of Methods*.
- Patel, Amit. “Introduction to the A* Algorithm.” Red Blob Games. June, 2016. https://www.redblobgames.com/pathfinding/a-star/introduction.html.
- Rossetti, Rosaldo and L. Ferreira and L. Teófilo and J. Filgueiras and F. Andrade. *Algoritmos em Grafos: Caminho mais curto*. Faculty of Engineering of University of Porto. Available at: https://paginas.fe.up.pt/~rossetti/rrwiki/lib/exe/fetch.php?media=teaching:1011:cal:05_2.06_1.grafos2_a.pdf.
- Sanders, Peter and Dominik Schultes. *Transit Node Routing based on Highway Hierarchies*. Karlsruhe University. Available at: http://algo2.iti.kit.edu/schultes/hwy/newYork.pdf.