TOPOLOGICAL ANALYSIS OF MULTIMODAL TRANSPORTATION NETWORKS

DATA STRUCTURE AND ADVANCED ALGORITHMS

Daniel Garcia Silva
up201806524@edu.fe.up.pt

Diogo Oliveira Reis
up201405015@edu.fe.up.pt

Luís Miguel Afonso Pinto
up201806206@edu.fe.up.pt

Sara Sicic
up202111383@edu.fe.up.pt

July 8, 2022

ABSTRACT

In order to analyze the topological structure of a transportation network from the multimodality perspective, research was conducted on the pedestrian, bus, and metro lines in the city of Porto. To achieve this, the algorithms were developed using "go" as the programming language. These algorithms include Tarjan algorithm, in order to measure connectivity along with Dijkstra, A*, and genetics-based metaheuristic algorithm in order to measure the shortest path between nodes and also the cheapest path. To build a single multigraph using data from three different sources, a way of consolidating all this divergent data was developed by using a two-dimensional K-D tree. The purpose of this was to merge all the disconnected graphs generated from the three different datasets into a single connected multigraph. Finally, and in order to visualize the path obtained, a multigraph library named "NetworkX" was used and later improved upon using fogleman GG to draw the tiles and Leaflet to visualize the graph.

Keywords Topology Analysis · Multigraph · Transportation · Multimodal Network · Path Search

Contents

1 Introduction

As it is known, to be efficient and fair, transportation systems should serve diverse demands. For example, it would be inefficient if inadequate sidewalks and paths force parents to chauffeur children to local destinations to which they would rather walk or bicycle, or if inadequate mobility options force urban commuters to drive although they would prefer to use public transit, such as buses, trams, and metro. Physically, economically and socially disadvantaged people, in particular, need diverse mobility options: walking and cycling for local travel, public transit for longer trips, and private vehicles when necessary. The goal of this project is to analyze the topological structure of a transportation

network from a multimodality perspective. To achieve the multimodality of the transportation system, the system should be properly examined. Public transportation poses the biggest problem of effective transportation, so it was decided to examine bus and metro lines in the city of Porto in this project. To access all the functionalities developed, a menu was implemented in order to facilitate usage. The menu offers the following options:

- Run setup
- Perform the connectivity analysis
- Find the shortest path between two nodes
- See the map
- Load raw
- Export
- Exit

2 Literature review

Regarding the work that was used as a basis for the project, there are a few examples of papers that describe useful algorithms for the use in graphs that represent city's roads and public transportation networks.
There are existing platforms that can be used to search for paths between two points that optimize for time rather than distance such as Google Maps or Michelin. However, there are some drawbacks to each of these platforms. Michelin has zero to no integration of public transportation networks. On the other hand, Google Maps has the ability to integrate bus and metro in the search but only allows for time optimization, rather than a price, which we have included as a feature in our project.

3 Methodological approach

This section will discuss the many approaches to the problem in a more broad approach, as well as the solutions to the many problems found during the exploration of the chosen approach.

3.1 Dataset and its treatment

3.1.1 Dataset Sources

The first problem we found was the need to find suitable datasets for our problem. We chose to focus on one city/metropolitan area. Since we are students in Porto, we decided to use the data available for our zone mainly the Metro Network and the bus/tram network of STCP, both operate mainly in Porto and Vila Nova de Gaia but extend in length across the AMP (Área Metropolitana do Porto).
The bus data was provided to us by professor Rosetti, and metro data was downloaded from the Metro do Porto website. The road data we acquired through the OpenStreetMaps database.
The bus data is stored in two .csv tables. The first one, lines.csv has two columns which include the codes and names of the bus line depending on its direction. The second bus data file is stops.csv, and is composed of columns: Code, Name, Zone, Latitude, and Longitude.
The metro information is composed of many files. The main ones are the stops.txt which specify the information about each stop such as id, coordinates, and full name. The stop_times.txt file was used to extract the lines and their schedule. This file contained information about the arrival and departure of the train, the stop id of the specified stop, and each line had the information about one stop in time for each line.

Having this information, the roads information to connect the two networks and allow for walking between stations was missing. To fix this, we chose the publicly available information of OSM (Open Street Maps) which contains information given by thousands of contributors across the world. We used a stable version, downloaded from Geofabrik that keeps daily updates of the maps.

The result was a dataset with:

- Porto Map: 283385 nodes, 559419 edges
- Bus Map: 2487 nodes, 5012 edges
- Metro Map: 82 nodes, 105 edges

3.1.2 Chosen programming languages

The algorithms and data structures were developed using "Go" as the programming language. "Go" is an open-source programming language that is based on "C" language and was developed by Google in 2009. The programming language comes loaded with strict typing, memory management as well collection, and is built to manage concurrency. It is leveraged by leading brands across the globe such as Dropbox, Twitter, Uber, and many others, even though it is a comparatively new language. "Go" has a good standard library, built-in concurrency mechanisms, has good performance comparable to languages like "C" and a good learning curve and development time. It also possesses built-in mechanisms for XML and csv file reading, useful for data management tasks. Furthermore, the group members were interested in learning a new language.
Alongside "Go", "Python" was also used to implement the graph visualization task. This is further explained in a later section called Visualization.

3.1.3 OSM Dataset Treatment

The information from the Metro as well as the Bus information was concise and had only the information that we needed for the project. However, the data from the OSM had much information that was not needed in our context. Information such as administrative boundaries, buildings, electricity lines, water pipes, and so on. Also the information extracted from Geofabrik had information regarding Portugal and not only the AMP. This information summed up to the final size of 5.39GB which we found to be excessive.
The OSM data can be in one of three types defined in the XML.

1. Node
2. Way (e.g., paths and house boundaries)
3. Relation (e.g., administrative boundaries, electrical lines)

Both 'Way' nodes and 'Relation' nodes are a collection of 'Node' ordered in a logical way. The path nodes here are ordered as they appear in the given path. The 'Relation' nodes were excluded as they did not offer useful information. Our first approach was to crop the data to match our Metro and Bus Data. We calculated the boundaries of those stations and cropped the data the nodes that fall outside that area (with a 5% margin outside the boundaries).
We also removed every node that did not have the tag 'highway'.This tag is used to mark nodes that are connected to paths, from a rural path only suitable for people to high-speed motorways.
Excluding every other node, allow us to keep only the nodes regarding the paths, which were the ones that were relevant to us. After having extracted the nodes, we eliminated every way that contained nodes that were removed.
This posed a problem as we were removing ways that were the only connection for some nodes just because that way had referenced nodes. Most of those nodes would become unconnected components of the graph, so we removed every node that was not connected to any other node (inbound or outbound).

This part was a manual step that involved an approach tailored to the specific data we had in hand. The following steps can be seen as more generic and easily used in other areas than Porto.

The result of this filtering was stored in a file that was used in further steps.

3.1.4 General Dataset Treatment

After we had extracted the information from the multiple sources, there was the need to connect them and make them useful as a whole, for that we needed to import them. We chose to represent the data as nodes and edges in a directed graph. Where the stations, intersections and POIS are the nodes and the streets, paths, tracks and bus lines are the edges of the graph.

The nodes were chosen to be the same no matter their source. They were named differently, so we could distinguish them when debugging.

The nodes were loaded into a specific structure:

This structure keeps track of inbound edges (used when removing a node to check which nodes reference a node) as well as outbound edges. It also keeps the coordinates information, the name of the stop, its zone and an internal unique identifier, independent from the dataset ids. Also, for further use there are two flags that keep track if a node has been referenced before, used in search algorithms and a flag to check if it is a station, to be able to distinguish stations from ordinary graph nodes (intersections and nodes used by OSM to simulate a curve with many straight lines).

Edges were calculated during setup, and their weights were calculated from distance (straight line distance in OSM data) into time. The bus and metro info already had the wight as tie so we kept it.

An edge type was made so we could distinguish between metro, bus and road connections, useful for the visualization part.

3.1.5 Turning the Datasets into a coherent graph

After we had cleaned the datasets and loaded them into a graph we needed to connect the three separated graphs. Metro Road and Bus. That information was loaded into the same graph but there was nothing that would connect those three networks. Our first idea was to use the stations nodes found in the OSM data and match them using coordinates. However, those nodes were not complete and their coordinates did not always matched the coordinates from the other sources.
We kept the idea of using the road network (the most extensive of the three networks) as the glue to unify the networks, but we changed our approach to connect the Metro and Bus stations to nodes in the OSM part of the graph. We decided we would find, for each station, the closest node in the OSM network and create a fictional connection between those two nodes. This way we connected the Bus stations and network to the road network and connected it with the metro network.

Our first idea was to use a naive approach were we would, with a linear search, find the second closest node to the station (the closest being the node itself). This proved to be a bad approach due to the enormous time the search would take since it had to be repeated for each station.

Our second approach was to use a specif data structure that allows for fast geographical search. Geographical search is often a problem due to its nature. Instead of being one dimensional, like numbers, its has many dimensions (2 or 3) which makes the search more difficult.

From the analysed algorithms we decide to use the kd-tree(2d-tree in this case) to allow us a fast search for the closest node to each station in the graph. A K-D Tree was implemented to speed up the load of the network. A K-D Tree, short for k-dimensional tree, is a space-partitioning data structure for organizing points in a k-dimensional space. K-D Trees are a useful data structure for several applications, such as searches involving a multidimensional search key, like range searches or nearest neighbor searches, and creating point clouds. K-D Trees are a special case of binary space partitioning trees. Every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with a larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x value of the point, and its normal would be the unit x-axis. The K-D Tree implemented was a two dimensional K-D Tree.

Further improvements were made by using a concurrent creation algorithm for the tree, allowing for four threads to be running at a time. This was done by using the way the trees are made which can be divided as independent recursive running functions allowing for multi-threading. After this step was done we had a connected graph and we were able to proceed with the search for paths in the graph..

Final dataset:

- 156 341 nodes
- 334 534 edges

3.2 Connectivity

The connectivity analysis was made after the graph was fully loaded and the stations connected with the road network. This proved to be a poor decision, corrected after we realised Its consequences.

We moved this analysis to be run before the conjugation of the three networks, so we could further improve the road network. The main problem was the Strong vs Weak connectivity. Our filtering was only about weak connectivity, we check if the nodes were connected to any other node. While this is acceptable in a undirected graph, in a di-graph it may exist some situations where there is a path from A to B but not form B to A making the node A isolated in practice. What the connectivity analysis showed us was that there where hundreds of points in that situation and also points that where in small connected components. Apart from the main component (297345 node) the next big component had 23 nodes. This is due to the source of the data, OSM user cannot always provide the best data and sometimes odd thing happen in the data.

The main issue with making this analysis and filtering after the 3.1.5 section was the existence of some points that would connect to some nodes that would belong to small/not coherent sets. We after moved this entire section to before the work done in the section 3.1.5 and those problems were rectified.

Another solution we discussed would be to connect those disconnected components to the other components, but we decided that the elimination of such nodes would be a better approach due to Its simplicity and the fact that the creation of such connections would be extremely inaccurate.

We chose to use the Tarjan's algorithm. It allow us to fetch not only the number of SCC(strongly connected components) but also their nodes as well as the WCC (weakly connected components). With this we were able to analyse the best approach to the connectivity problem (keep the status quo, eliminate not connected components or connect components).

We were also able to conclude that the Bus and Metro networks are each one in a SCC,(which makes sense in our context) and the road network is not on SCC, in fact it consist of 233 SCC. This was a more difficult to understand fact. Since every node in a network should have an in and out way. If we can get in we have to be able to get out, and if we can get out it make only sense we can get in.

As we discussed previously we assumed this problem comes from the, somewhat inaccurate source of data (OSM).

This module was also made available to run outside the regular setup algorithm.

3.3 Setup Save

After we filtered and treated the information we gathered, the graph was stored so we could easily reuse it in future runs of the program since this steps took substantial time and we could in fact save the work previously done.

3.4 Shortest Path

As discussed previously, the edges weights were already calculated to be time in seconds the edge takes.

Another think we should assure before we start to look for the shortest Path between two points is to make sure we are able pin point a starting an ending point for the journey. As our console input are coordinates, we decided we would take a similar approach to the one we took in stations situation. We take the point and we try to find the closest point to the specified start or ending point. Once the point is found, it is then considered the start/end of the trip.
We decided to make this by using the same 2d-tree approach. The naive approach would be ok if we consider that this would only be done twice per path finding query, however, if we had already the 2d-tree created we would actually be able to use it to speed up the search. As the creation of the tree was a one time thing per run of the program, we decided it would compensate the time spared in case multiple instances of the path finder algorithm were run.

One of the first things we decided was that the total distance of the trip should not be something we would offer as a possible query. The available modes of transportation make the distance almost irrelevant. 3 kilometers may represent 1 hour if by foot, but 5 minutes by metro, and so the result would be meaningless. Instead we focused on getting the shortest paths regarding time.

3.4.1 Shortest Time Path

The biggest problem in our hands, after we were able to create a coherent graph, was the ability to query in real time the shortest path between a pair of points in terms of time. Our first approach was to use the simple Dijkstra algorithm and improve from there. Form the beginning we also had a preoccupation in using the most adequate data structures for the job. In a Algorithm such as Dijkstra a crucial part is the queue used to sort the candidates for exploration, a slightly slower algorithm could lead to huge time delays in the Dijkstra algorithm.

Using a fast heap algorithm for priority queues improves the asymptotic running time of important algorithms, such as Dijkstra's algorithm for computing the shortest path between two nodes in a graph, compared to the same algorithm using other slower priority queue data structures. Since we were planning to use Dijkstra's algorithm and other algorithms as it will be explained later, this was a crucial data structure to implement, in order to improve efficiency.

After exploring the available priority queue algorithms we chose to go with the Fibonacci priority queue that allows for constant insertion and decrease-key, essential for the algorithm. Another approach studied was the use of a simple priority queue with a worse Decrease-key time. With this approach we would mimic the decrease of a key by inserting a new value and keep track of the nodes explored. If a node was to be explored a second+ time, that would be ignored. This way we could ignore the slow decrease key time and only depend on the insert time.

A second approach was chosen to tackle the shortest path algorithm. The use of an heuristic that would speed up the process by guiding the search, instead of a radial search, typical in Dijkstra. This works by adding an estimation of the shortest path between the current point and the destination, preventing the cases in Dijkstra where there are nodes being explored that are too far way from the destination to be viable.
It was only after we had implemented the A* algorithm that we found that it would not be suited for our case. Our implementation of Dijkstra was actually the same of the A*, the only difference was the function used as heuristic.(return 0 if Dijkstra or distance/speed in a*)
As we were using time as our weight, we needed to calculate the estimations based on time, rather than distance in straight line. For that we used the straight line distance and divided that by the speed of circulation. And this speed was the main issue with the A* approach. The existence of three modes of transportation (by foot, bus and metro) was also the reason the speeds were so inconsistent. To try and achieve a optimal result we had to use the metro speed to calculate the estimations. This lead to hugely optimistic estimations (most of the times the difference where superior to 40 times) that made the A* behave almost as Dijkstra making no improvement on the query time.(Sometimes queries were actually worst due to the heuristic function overhead).

Other implementation, based on A* were studied before we found the problem of A*. The main one was the use of Anytime A* combined with weighted A*. This works by multiplying the heuristic by a weight that will lead to over-estimations. This makes the heuristic not acceptable and so the results would lead to sub-optimal paths. This results are, however, bounded by the weight. A weight above one will lead to sub-optimal results, a weight of one is the same of A* and a weight of 0 is the same of Dijkstra.
By running this algorithm with a relaxed heuristic we achieve better time results, at the expense of optimal results. By decreasing the weight we can progressively decrease the errors until we get an optimal result (weight = 1) or the max time is reached. This algorithm allows us to reach a result at anytime, even if not optimal.
The higher the weight the closest the algorithm is to a pure greedy algorithm. The problem here was the same of the ordinary A*. Our base estimations were too optimistic (considering every road could be cruised at metro speed) that the results were either too bad to be useful, or too close to Dijkstra times to make sense to use them.

Further approaches were taken into consideration, mainly the introduction of metaheuristics as well as ALT for faster, yet close to optimal, solutions.

3.4.2 Improve query times by pre-calculating some route times

After we reached the disappointing results of A*, we needed a way to improve the query times. Our approach was to pre-calculate the shortest distance between stations by foot and add those results as edges between the stations. After each query we would reconstruct the original paths by using Dijkstra. As this second step was made on smaller scale than the original query, its times were much lower than the original ones. This second step was needed to allow for the user to see a real path, rather than an artificial one created to speed up the results. We could have saved the paths as well as the times but that would lead to huge storage space.

For that we needed to know the distances between each pair of stations. For that we had two main approaches, use Dijkstra implemented before, and run it for each station, the stopping criteria needed to be changed to allow for the algorithm to search for the Shortest time between a station and every other station, rather than between two stations.

The other approach would be to use the Floyd-Warshall algorithm.

The first was chosen due to time complexity issues as stated below. The main issue was that we only cared about distances between every station. While Dijkstra would give us the distance between each station and every other point, Floyd-Warshall would give the distance between every pair of points, stations or not.

3.5 Visualization V.1

The last step of the work was assuring that it was possible to visualize the multigraph. This was a difficult task because of the enormous amount of data. In order to accomplish this visualization task, "NetworkX", a python package used for the creation, manipulation, and study of the structure, dynamic, and functions of complex networks, was used. Furthermore, the MultiGraph class already is implemented in this library. Due to the large amount of data, only a portion of Porto city was able to be computed in a reasonable time. See Figure 1. In this visualization, the walk nodes and edges are represented by the color green, the bus nodes and edges are represented by the color blue and the metro nodes and lines are represented in orange. The path obtained as a result of section 3 is represented in black.

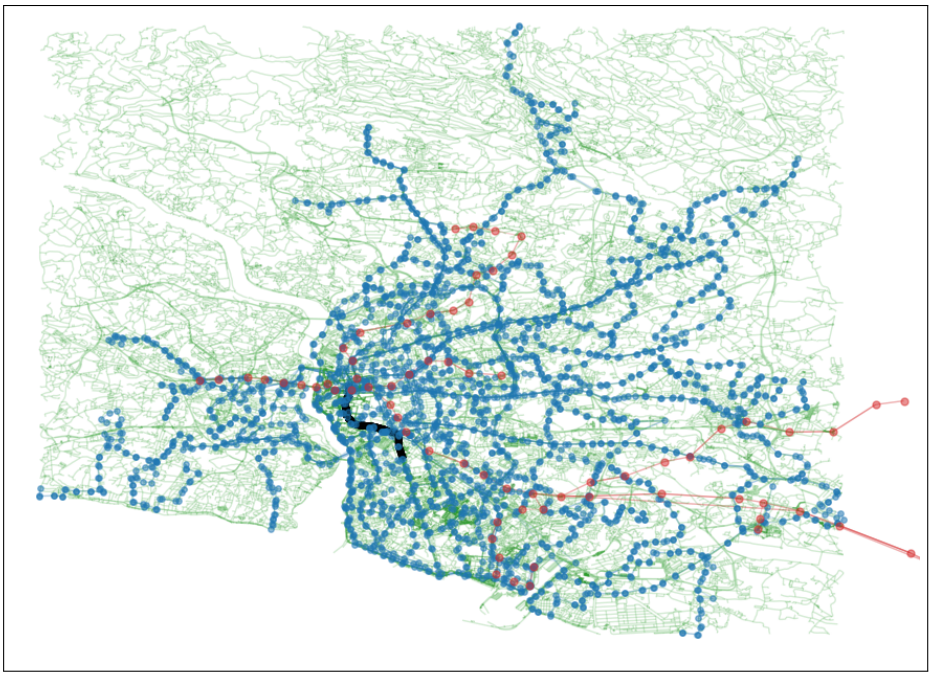

Figure 1: Path visualization version 1

3.6 Visualization V.2

Since the original implementation of the visualization was slower than what we needed for the purposes of the project, some changes were made in the visualization part. Quadtree data structure was used in order to faster retrieve the tiles. In the new version of graph visualization, the whole graph doesn't have to be drawn, only the specific tiles do. The graphical user interface was also updated to be more clear and accessible. In order to visualize the graph faster, we replaced the NetworkX package with fogleman GG to draw the tiles and Leaflet to visualize the new graph. The interface now supports for the user to select the source and the destination of the search by clicking on the map. Afterwards, the user can select one of the implemented algorithms for calculating the shortest path by pressing one of the buttons. Following this a red line representing the shortest path between the two points is drawn. This approach to the visualization has in the end proven to be faster and easier to use. 2

3.6.1 Interactive Map

Another important addition was the ability to use the map as an interactive part of the project. The path can be chosen using just clicks instead of inserting manual coordinates. For this, the index.html file was added. Also, the main algorithms can be performed just by clicking on the buttons in the page. 3

3.7 Usage

3.7.1 Setup

The menu is quite straightforward to use. It allows for the Setup of the graph, where the costly part is done. It saves the result for further use.

3.7.2 Load

The load part loads the result of the setup

3.7.3 Connectivity Analysis

Shows the result of the Tarjan's connectivity algorithm

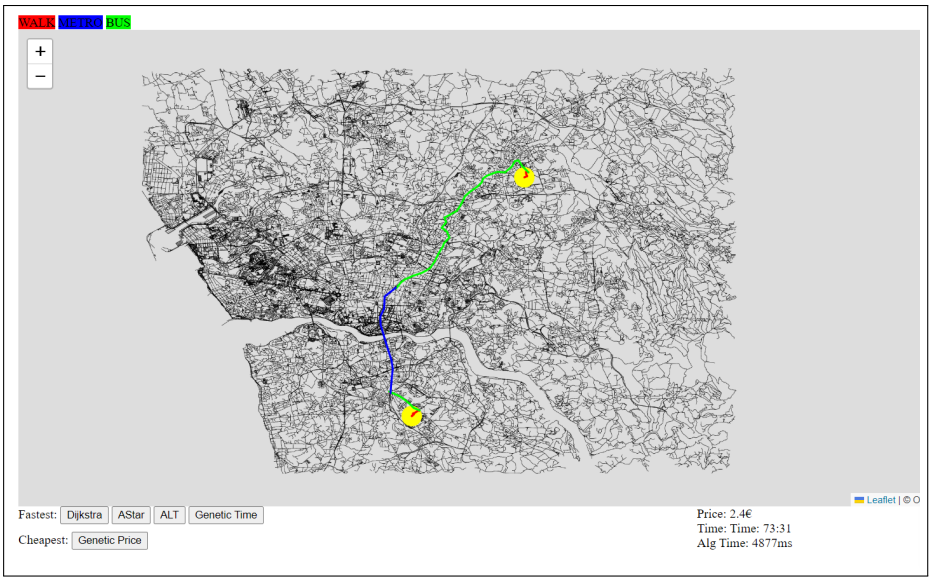

Figure 2: Path visualization version 2

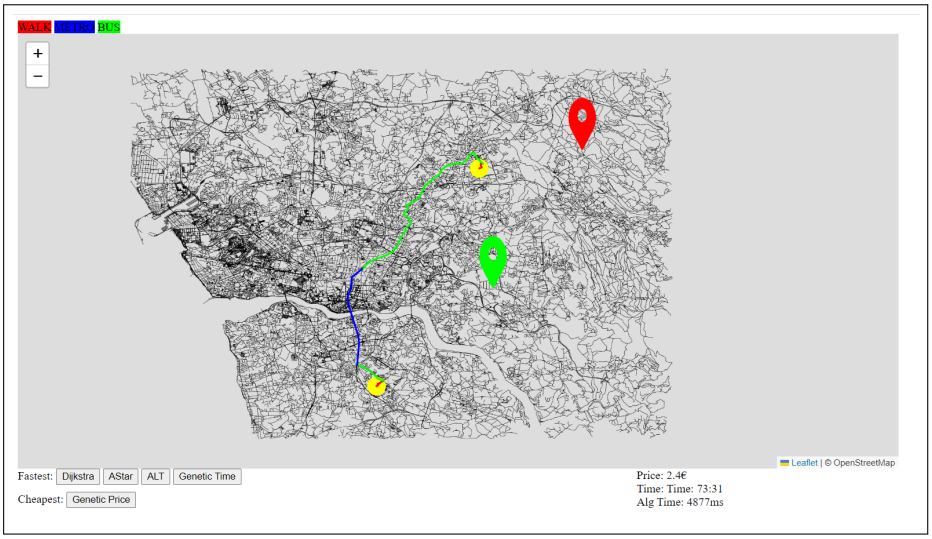

Figure 3: Choosing path and algorithm v2

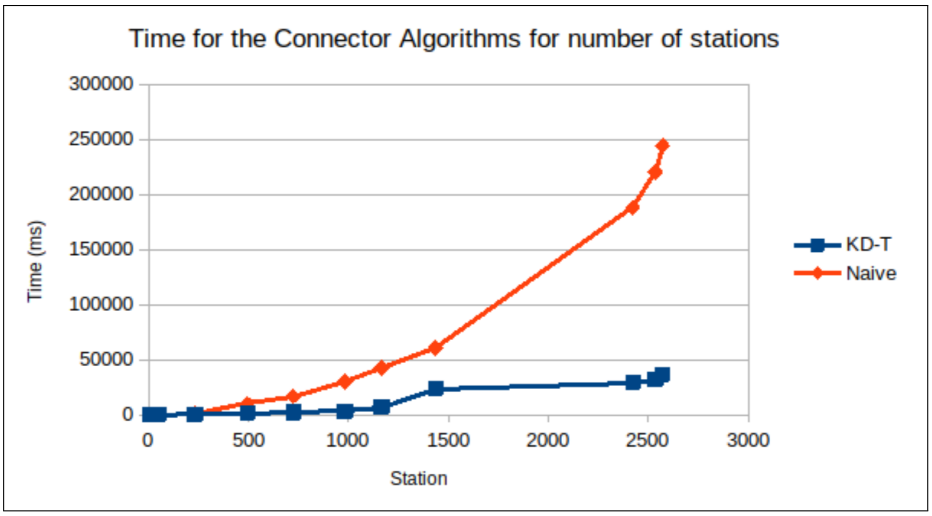

Figure 4: Time for the connector algorithm by number of stations

3.7.4 Find Path

Allows to search for the shortest path between two points It is possible to input the start and ending point as a pair of coordinates:

lat,lon

If enter is pressed the point used will be a random one.

3.7.5 See Map

The map will load, may take some time 20s+. The program is usable while the map is being loaded. This load time was the main reason why visualization version 2 was implemented.

3.7.6 Load Raw

Loads the info without filtering and network connection

3.7.7 Export

Exports the current graph to be used by load function the next time the program is started

4 Results and Discussion

4.1 Data structures

In this subsection we will be discussing the chosen data structures for the multiple problems presented above

4.1.1 2D-tree

The choice for the 2d-tree came after we tested the algorithm to connect the three networks. The first approach, linear search for the closest point for each station proved to me quite inefficient. Even though there was no need for setup before we could run the algorithm(unlike 2d-tree) the time complexity was of O(s*n), with s = stations and n = nodes in graph We after went with the implementation of a 2d-tree over the linear approach, orders of magnitude slower. For that we needed two processes, the creation of the tree and the actual query. The complexity for the creation was of O(n*log(n)) while the had an average complexity of O(log(n)) with the total complexity being O(n*log(n) + log(n)*s), with s = stations and n = nodes.

This second approach proved better in terms of performance and we decided to keep it even at the expense of space being taken by the tree. See Figure 4.

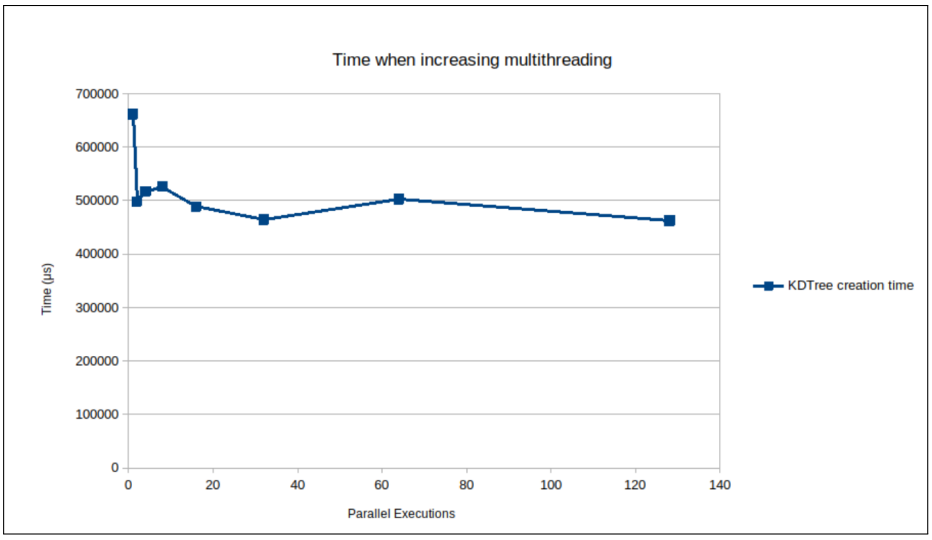

Figure 5: Time for the connector algorithm by number of stations

Further improvements were made by using a concurrent creation algorithm for the tree, allowing for four threads to be running at a time. This was done by using the way the trees are made which can be divided as independent recursive running functions allowing for multi-threading.

The results however of the multi-threading were not completely good as the time stabilized after four concurrent threads. We can, however see a major improvement between a sequential running and the following concurrent runs. See Figure 5.

This tree could be deleted after the algorithm was done but we decided to keep it so we could improve the performance of the shortest path algorithm that also needs to find the closest node to a certain coordinate.

As for the space complexity it has a O(n), with n = nodes in graph because it keeps a copy for each node in the graph.

4.1.2 Fibonacci queue

From the beginning we decided to use a Fibonacci queue to improve the performance of the algorithms such as Dijkstra and A*.
We had however some doubts if the use of the said queue would improve the performance. The theoretical analysis of the algorithm suggest that; however a second approach was studied with a simpler algorithm, use a simple priority queue with linear decrease-key time and instead of keeping decreasing the values when they needed to be updated, we simply added a new value. We also kept a flag on the node to check if a specific node had already been explored. Since we were using pointers, the first time a queue_node appeared on the top of the queue, its real node would be tagged as explored. The next time that node appeared on the top of the queue, with a different value, we could check if the real node had already been explored. With this approach we could simplify the process by eliminating the huge complexity of a Fibonacci Queue.

Our results were not conclusive as we could not identify the best approach with an empirical evaluation as the results were mixed. We finally decided to use the Fibonacci queue for the final code.

4.1.3 Quadtree

We decided to use Leaflet for the improvement of the visualization part. This needs the ability to fetch images of specific squares of the map. Unlike the kd-tree, the quadtree is seen as memory inefficient, however the use of it would allow us to further improve the visualization speed of the map. The problem we had in hands was the problem of finding every point inside a specific square of points, know by the Leaflet as tiles. This is exactly the kind of problem quadtrees aim to solve. By mapping each tile to a square of the tree, we were able to know the points of said tile. This however was a difficult task due to the memory problem. Two approaches were considered, the first one would be to keep a list of points in each node would easily fetch its nodes. This was impossible due to the memory limitations. The second approach was to keep only the nodes on the leafs and make the node search for its nodes recursively in its children, this would be great memory-wise but would delay the algorithm of tile drawing.

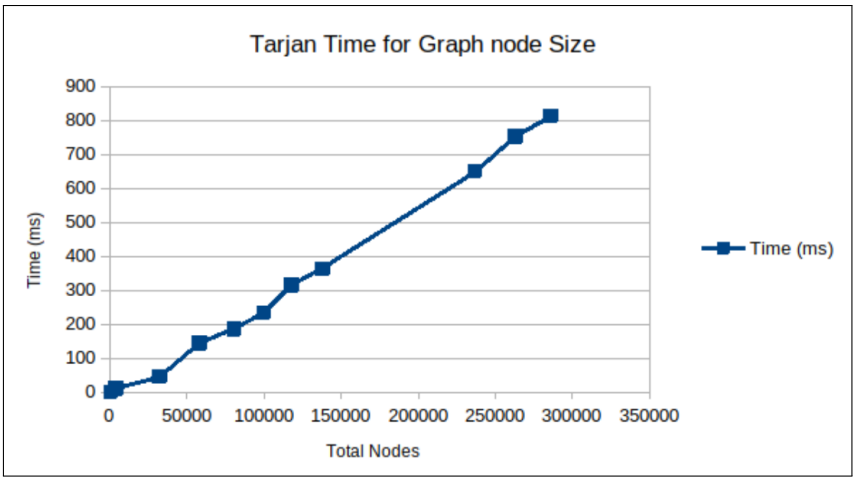

Figure 6: Time for the connector algorithm by number of stations

To solve the problem we used an hybrid approach. Each node/leaf keeps two integers that refer to the first and last point they have. This integers are the index of those points in a list of every point. The way this list is ordered can be seen as DFS. The first points on the list are the points of the first leaf, then the second, and this allows for each non terminal node to keep also those two integers. As its nodes are all in a row in that list, the only thing we need to fetch a node's points is their limits (first point and last point index) and then we can use those numbers to get a slice (as it is known in GOLANG) of the original list.

This allows for space efficiency, only 32 * 2 bits per node plus the nodes their-selves in a central list, and time efficient as one node does not need to go through its descendants to discover the points it stores.

4.2 Algorithms

4.2.1 Tarjan Algorithm

The tarjan algorithm as described in the original paper, relies on the fact that, on a directed graph, the existence of a cycle implies that the nodes inside that cycle are in the same Stringly Connected Component (SCC), that can be defined as

$\forall A, B \in G, existsPath(A, B) \wedge existsPath(B, A)$

This allowed us to retrieve the SCC where we could operate.

The algorithm is of the type DFS and so, it passes once per node, having a time complexity of O(n), with n =nodes in graph. As expected, the empirical analysis corroborated the theoretical time complexity. See figure 6

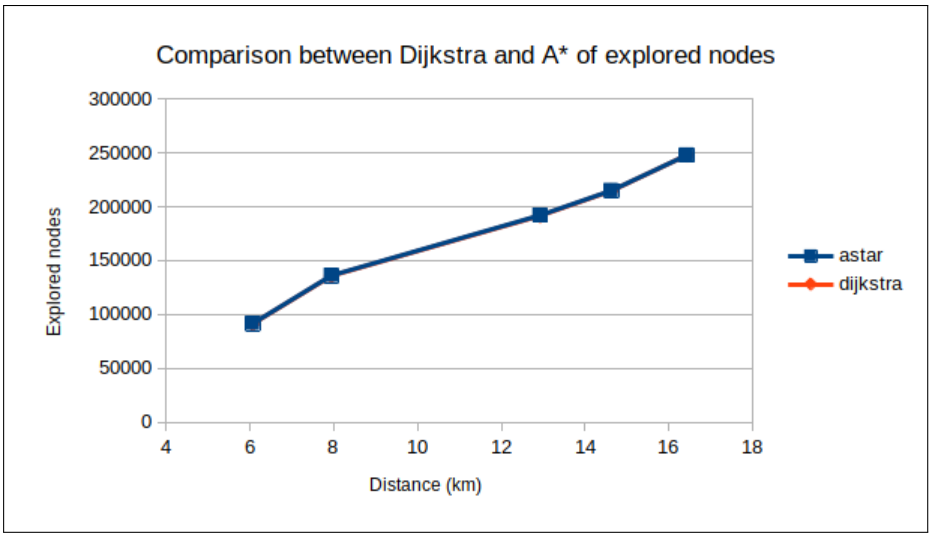

Figure 7: Comparison between Dijkstra and A* for explored nodes

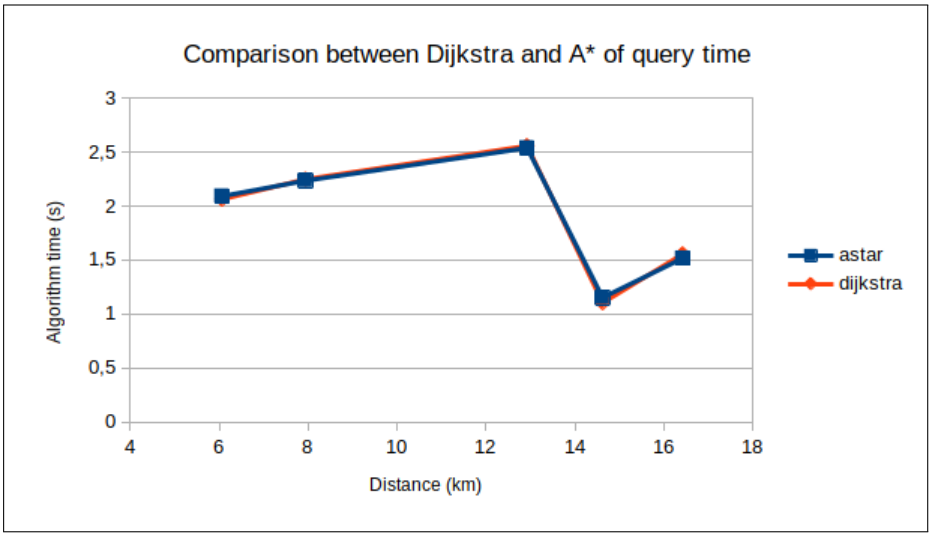

Figure 8: Comparison between Dijkstra and A* for the time spent

4.2.2 Dijkstra and A*

The Dijkstra can be seen as a specific case of the A* algorithm, where the heuristic value is always 0. Taking that into consideration, and ignoring the overhead of calling a function (would not be needed in a Dijkstra implementation) we decided to make the A* implementation the same of the Dijkstra. The heuristic is calculated using a function passed as an argument. If we want the Dijkstra we make the function return 0, otherwise we use the haversine function to calculate the distance between the two nodes (current and final) and divide by the speed of the most optimistic approach, metro speed of 33 km/h according to the calculations made with the real schedule.
The necessity for this value was explained before. Taking that into account we obtained the following values for time spent in the search and for the number of explored nodes. See figures 7 and 8

As expected the number of nodes explored by the A* algorithm is lower but not to a big margin. This might explain the almost similar time performance due to the overhead introduced in A* by the haversine function.

4.2.3 Dijkstra vs Floyd-Warshall

For the calculations between every station, needed to improve the performance we implemented both the Dijkstra for every station and the Floyd-Warshall algorithm.

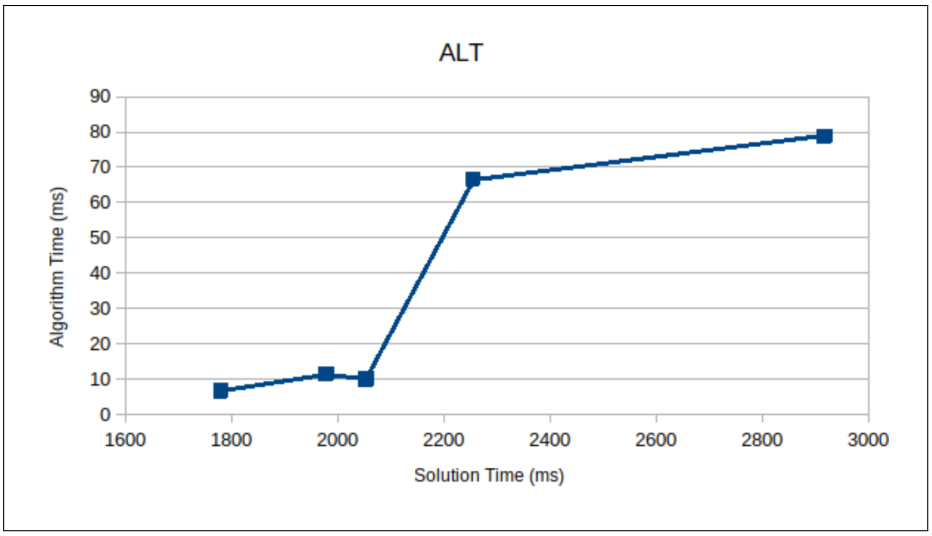

Figure 9: Worst case Time complexity : $O(V^2)$ (same of A*, hopefully better at most cases)

The Floyd-Warshall algorithm has an expected time complexity of $O(V^3)$ and the Dijkstra for each station has a time complexity of O(V*E*log(V)), with S = stations, E = edges and V = vertices.
The first analysis is that, if the E value is much smaller than the number of vertices squared, the Dijkstra approach is actually better than the FW algorithm. To support Dijkstra even more is the ability we have to restrict the number of Dijkstra runs from V to stations since we are only interested in the distances between stations for our optimization.

The Dijkstra final time complexity would be O(S*E*log(V)) we must assure that S*E is much smaller than V^3 and the Dijkstra approach is a more viable one. Since S is a small subset of V and the number of edges is in the same order of magnitude of the Vertices, we can conclude that S*E is much smaller than V^3 and so S*E*log(V) is less expensive in terms of time than V^3.

The analysis of this algorithm was somewhat impossible to do empirically since the times that the FW algorithm took escalated quite quickly. The Calculation of the Dijkstra showed a final time of around 50 minutes, using an S of 2569 nodes. The FW run out of memory after 3:32 minutes making us conclude that the Dijkstra was a better (and the only viable among the examined) solution.

4.2.4 ALT

In order to attempt to improve the heuristic function for the A* algorithm, the ALT (A* search, landmarks and triangle inequality) algorithm was used. This algorithm entails selecting a number of nodes (landmarks) and, for each of them, preprocess the distance from and to every other node.

Assuming a landmark L and nodes s and t, due to triangle inequality, we know that d(L, t) - d(L, s) <= d(s, t), or d(s, L) - d(t, L) <= d(s, t). This allows for the use of the maximum between these two differences as a heuristic. For more landmarks, simply use the maximum between all of them, since all of them are feasible.

In this implementation, the selection of landmarks was done manually. Taking into account that landmarks that work best are before or after one of the nodes[8], 12 nodes around the edges of the map were chosen, at roughly equivalent distance from each other.

In addition to this, instead of preprocessing every distance to and from every other node, only the distance to every other node was calculated and stored. This was done in order to save preprocessing time, and is justified since, apart from very few bus trips that do not match entirely in both ways, practically every edge from node A to node B has its equally weighted edge from B to A. The time saved by this decision is more than half of this preprocessing step, since it's only necessary to run 12 worst-case Dijkstra's and avoids running 12 Dijkstra's for every node in the graph.

ALT also entails picking a few active landmarks for each search, in order to reduce even further the constant heuristic calculation time. This implementation selects 3 active landmarks using the same method as the one described in the original paper that proposes this algorithm[9], which is to select the maximum landmarks between the source and destination nodes, and using those 3 for heuristic calculation on that search. This reduces the constant time of heuristic calculation to 1/4 of the time.

4.2.5 Genetics based metaheuristic for shortest path

The problem consists of finding the path that minimizes a metric such as distance, time, cost etc. to go from one node to another in a directed or undirected graph. Although standard algorithms and techniques such as Dijkstra and integer programming are capable of computing shortest paths in polynomial times, they become very slow when the network becomes very large [7]. Therefore, we have thought about using meta-heuristics to solve the routing issue in road networks. Meta-heuristics are capable of copying with additional constraints and providing optimal or near optimal routes within reasonable computational times in large-scale road networks [7]. The proposed approach is a combination between genetic algorithm and an adaptation of variable neighborhood search. The algorithm starts by creating initial solutions using for example a double search algorithm. Then, enhance those initial solutions using variable neighborhood search. This allows for new regions in the search space to be discovered and theoretically improves with up scaling. The mutation part of the algorithm works by re-calculating random paths between two nodes on the graph. We chose two random nodes in our path, ignore the known path between them and re run the shortest path algorithm between those two points. This allows for a fast yet valid new iteration of the original specimen.

The biggest limitation of the solution was the, somewhat slow performance of the mutation step. This step was slower than it usually is in this type of algorithms, but the results were actually significantly faster than Dijkstra. The fact that we were calculation only small portions of the path, made the problem much faster than calculating the full path as the area of explored nodes was smaller due to the close proximity of the nodes, when compared with Source->Destination.

This fact was also used to calculate the original solutions that are used by the algorithm. Instead of calculating the path from A->B, we divided A->B into portions and calculated the path between those portions. The straight line between the points was divided into 4 sections, A,P1,P2,P3,B and the paths were calculated as follows.

- A->P1
- A->P2
- P1->P2
- P2->P3
- P2->B

This step was also paralysed to improve the speed of the algorithm, even though the sums of the times it took to find these paths was lower than the times of A->B.

After this, we created the paths used on the initial solution:

- A->P2->B
- A->P1->P2->B
- A->P2->P3->B
- A->P1->P2->P3->B

Even though, the solutions had redundant parts, that would generate equal results on the generation of the next generation, we chose to keep the number of individuals the same throughout the generations, so we created this initial four solutions. 10

4.2.6 Genetics based metaheuristic for cheapest path

The main issue with the time problem is easily managed as we don't usually care about the means of transportation and walking is usual.y bounded for being slower than the other methods.

A problem we had with the cheapest path was that the obvious choice of algorithms were not viable because the cheapest path is always walking, and the results were always 100% of the path walking. We had to make sure the bus and metro lines were being taken into account. For that we used a similar approach to the genetics based metaheuristic for shortest path.

Same principle as the genetics based metaheuristic for shortest path but using a different cost function. In this algorithm the cost function is determined by the real zone prices of the Metro do Porto and STCP. We also assign the price of 50 cents per minute of walking if the walking time exceeds 10 minutes so that the algorithm would not prefer walking to the other modes of transportation.

This allows us to base the solution in shortest path solution, therefore limiting the time/price trade-off, but at the same time allows the mutations steps to explore cheaper paths and the next generation step to favor cheapest paths, constructed on top of shortest path focused algorithms.

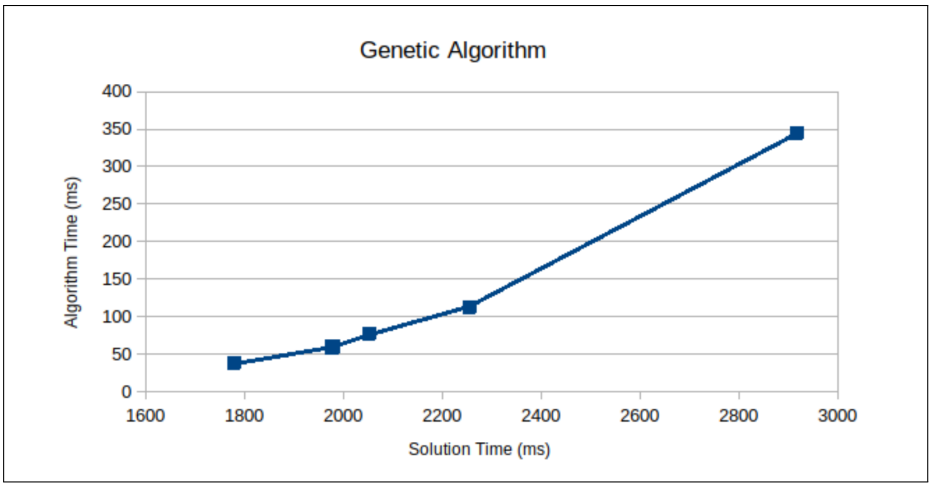

Figure 10: Worst case Time complexity : Hard to calculate, depends on the mutation algorithm used (A*)

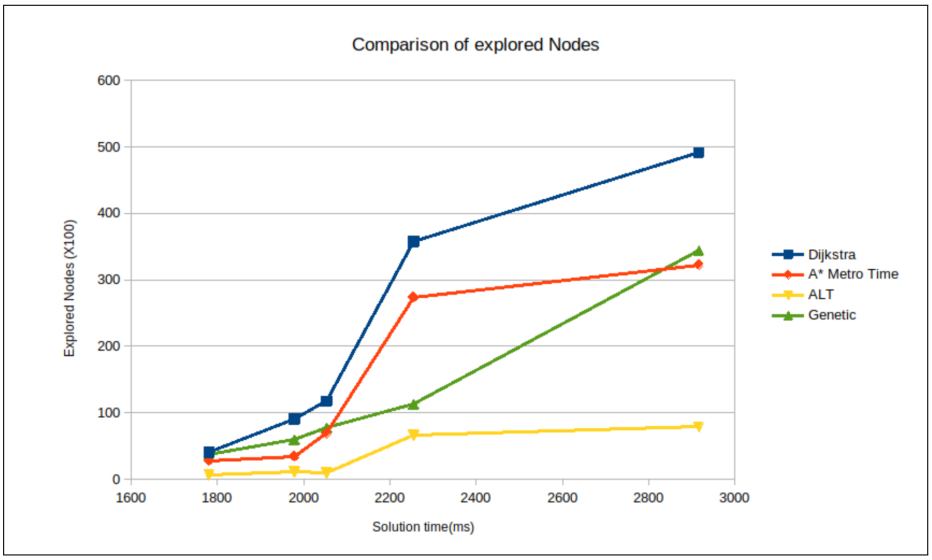

Figure 11: Comparison of explored nodes

4.3 Comparison between Dijkstra, A* with metro time, ALT and Genetics

In this section we compare the four algorithms discussed above, namely the comparison of explored nodes in 11, the comparison of quality of solutions in 12 and the time gain versus solution optimally in 13.

In order to evaluate the results 5 pairs of random start and end nodes and 7 measurements for each pair using each algorithm were used.

We can see, by analysing the results of said algorithms, that, ALT and Genetic Algorithm provide a good solution for problem, even though they are not always optimal. If we look into the Solution time, we can see in 12 that ALT actually provides the optimal solution in three of the five examples while Genetic is always worse than optimal. In picture 13 we can see the loss time of the algorithms when compared with the optimal solutions (ALT results are closer to optimal than Genetic) and also the time saved by the said algorithm. For instance, ALT takes only 27% of the time that Dijkstra uses while offering good results.

From this we can conclude, at least for our problem, that ALT is a great trade-off between execution time and loss in optimal results. Whilst, the genetic approach is more Generic and not specific for shortest path, and gives a good result but the time gained is not good when compared to ALT. This is mostly due to the problem added in the mutation phase, by using A* as the mutation algorithm which generates slow mutations.

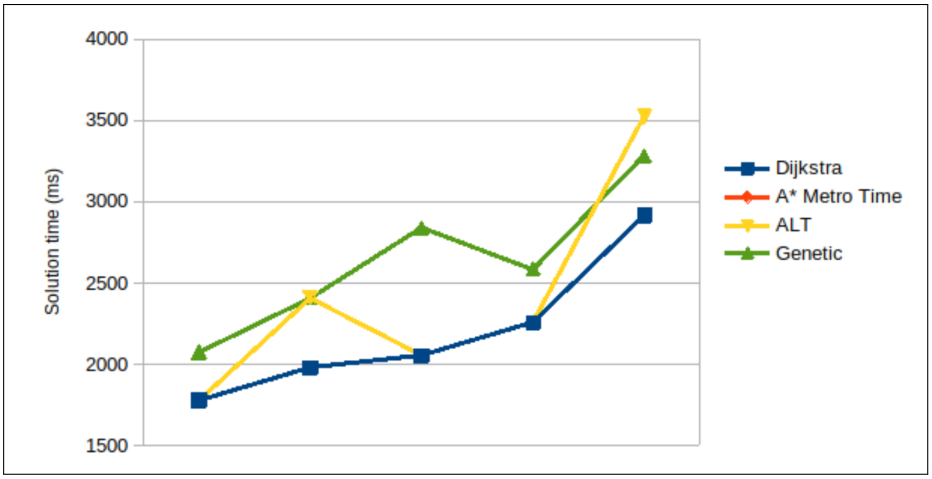

Figure 12: Comparison of quality of solutions

Average	A* Metro Time	ALT	Genetic
Solution Loss	0%	18%	22%
Time Gain	26%	73%	29%

Figure 13: Time Gain vs Solution Optimally

5 Conclusions

During this project we realized the theoretical complexities of algorithms but also how differently they can really behave in real world implementations. We originally encountered this with Fibonacci heaps, which were supposed to execute faster than the binomial approach, but ultimately ended up being quite similar due to its implementation complexity that added overhead and undermine its advantages over the other algorithms. Additionally, a good deal of times the approaches we thought would be straightforward and exact, ended up not being the most correct path, such as the A* vs Dijkstra situation where we should have a great difference between the algorithms but the data and its restrictions and idiosyncrasies made completely altered the way we had to look at the problem.
In situations where the time may be a constraint and, like in this case, the optimal solution is not always required, we can have other approaches that may actually lead to the optimal solution but at a fraction of the original time. This is particularly useful in problems where the time complexity is big enough for it to become impossible to solve such problem in acceptable/viable time. This made us be more aware of the possible errors one makes when evaluating a problem and also prepared us for a more "agile" approach to the problems to be able to cope with the perceived challenges of real life data, that often does not behave as expected. In conclusion, throughout the development of the project, we have found that it is often beneficial to try out more different algorithms and data structures since the final result is not always intuitive because of the different performance of the algorithms depending on the original data.

The work distribution was as follows: 45% for Luís Miguel Afonso Pinto, 25% for Daniel Garcia Silva, 20% for Diogo Oliveira Reis and 10% for Sara Sicic.

6 Future Work

From the previous sections we were able to identify multiple areas for improvement regarding the quality of life features. After examining the completed work, we have concluded that the further advancements could be made by improving the execution time with multithreading. Moreover, to improve the real-life accessibility of the project, the time to change the means of transport and the existent schedule should be included in the calculations. To summarize, the possible improvements include these areas:

- Improve time by using Dijkstra multithreading, adapting the bi-dijkstra algorithm, since A* didn't prove to be better than Dijkstra
- Use time more realistically by including transportation device changing time
- Take starting time into consideration since metro and bus schedules are not taken into consideration so far

References

[1] https://go.dev/learn/

[2] https://networkx.org/

[3] https://www.cs.cmu.edu/ 15451-f18/lectures/lec19-DFS-strong-components.pdf, visited in 02/05/2022

[4] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269–271.

[5] Maxim Likhachev. A* and Weighted A* Search. https://www.cs.cmu.edu/ motionplanning/lecture/Asearch_v8.pdf, visited in 02/05/2022

[6] https://wiki.openstreetmap.org/wiki/Main_Page

[7] Omar Dib, Marie-Ange Manier, Alexandre Caminada. A Hybrid Metaheuristic for Routing in Road Networks. 2015 IEEE 18th International Conference on Intelligent Transportation Systems, Sep 2015, Las Palmas, Spain. pp.765 - 770, ff10.1109/ITSC.2015.129ff. ffhal-01520124f

[8] Andrew V. Goldberg, Chris Harrelson, Haim Kaplan, Renato F. Werneck. Efficient Point-to-Point Shortest Path Algorithm in https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP

[9] Goldberg, A. V. and Harrelson, C. (2004). Computing the shortest path: A* search meets graph theory. Technical report.

Appendices

A Data for the given plots

Stations	Total nodes	Time	Time (ms)
2571	285956	812957537	812,957537
2535	263138	752649214	752,649214
2423	237005	650312001	650,312001
1435	138049	363493400	363,4934
1165	117899	315813305	315,813305
982	99693	233744980	233,74498
723	80318	187294262	187,294262
494	58085	145696569	145,696569
233	32184	46618971	46,618971
47	3853	10764968	10,764968
9	450	770090	0,77009

Table 1: Data for figure 6

Stations					Naive		
2571	285956	36341118655	36341,118655	244587289135	244587,289135	-208246,17048	0
2535	263138	32313196006	32313,196006	219783781843	219783,781843	-187470,585837	8
2423	237005	29417501609	29417,501609	187793276490	187793,27649	-158375,774881	4
1435	138049	23703225658	23703,225658	60866628500	60866,6285	-37163,402842	2
1165	117899	7280543478	7280,543478	42949347490	42949,34749	-35668,804012	1,9
982	99693	3382874345	3382,874345	30184832795	30184,832795	-26801,95845	1,8
723	80318	2455561751	2455,561751	16994307632	16994,307632	-14538,745881	1,7
494	58085	1401465181	1401,465181	10175352238	10175,352238	-8773,887057	1,6
233	32184	549836022	549,836022	856665284	856,665284	-306,829262	1,5
47	3853	73181872	73,181872	24797039	24,797039	48,384833	1,4
9	450	2994051	2,994051	404135	0,404135	2,589916	1,35

Table 2: Data for figure 4

Dijsktra			
Nodes Explored	Time (s)	Distance (km)	Time (s)
91409	1,104003	6,065122	2,06254188
135873	1,5561111	7,944116	2,24803592
191634	2,06254188	12,92675	2,55679428
214679	2,24803592	14,624502	1,104003
247813	2,55679428	16,426284	1,5561111

Table 3: Data for figure 7 and 8

A*			
Nodes Explored	Time (s)	Distance (km)	Time (s)
91648	1,15257948	6,065122	2,0914813
136186	1,51779524	7,944116	2,23503286
192058	2,0914813	12,92675	2,53736294
214912	2,23503286	14,624502	1,15257948
247821	2,53736294	16,426284	1,51779524

Table 4: Data for figure 7 and 8