

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona



Route planning for e-scooters

A Degree Thesis submitted to the Faculty of the
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Marc Gracia i Riera

In partial fulfillment of the requirements for the degree in

BACHELOR'S DEGREE IN TELECOMMUNICATIONS TECHNOLOGIES AND SERVICES ENGINEERING

Advisor: Marek Cuchý
Co-Advisor: Juan José Costa Prats

Prague, Czech Republic. Spring 2021



FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

Abstract

According to Wikipedia; The vehicle routing problem (VRP) is a combinatorial integer programming and optimisation problem that asks "What is the optimal set of routes that a fleet of vehicles should travel to deliver to a given set of customers?".

Many years have passed since Dantzig and Ramser introduced this problem in 1959. They described a real-world application concerning the delivery of gasoline to service stations and proposed the first mathematical programming formulation and algorithmic approach. But even so, route optimisation is now more important than ever. Large delivery companies invest a lot of their capital in VRP consultancy, knowing the most optimal route to deliver saves you a lot of time and money.

This project attempts to solve a route planning problem. This problem is based on the primary notions of a TSP (Travel Salesman Problem). Tries to solve a problem where we are in a city on an e-scooter and we want to visit a number of places in that city in the shortest possible time.

To reach all the places that are a goal for you, the battery of the e-scooter has to be considered, as it is decreasing through the distance travelled. So you have to consider if it is worth to deviate from the fastest route to take another e-scooter that has enough battery to reach the next destination, changing e-scooter adds extra time (time in which you change from one e-scooter to another).

The goal of the thesis is to design and implement an algorithm solving the problem, i.e. that shows you the fastest route and the time it takes to visit all destinations in the shortest possible time.

Resum

Segons la Viquipèdia; El problema d'enrutament de vehicles (VRP) és un problema combinatori de programació sencera i optimització que es pregunta "Quin és el conjunt òptim de rutes que ha de recórrer una flota de vehicles per lliurar a un conjunt donat de clients?".

Han passat molts anys des que Dantzig i Ramser van introduir aquest problema en 1959. Van descriure una aplicació del món real relativa al lliurament de gasolina en estacions de servei i van proposar la primera formulació de programació matemàtica i un enfocament algorítmic. Però tot i això, l'optimització de rutes és ara més important que mai. Les grans empreses de repartiment inverteixen gran part del seu capital en consultories de VRP, ja que conèixer la ruta òptima per al repartiment els hi estalvia molt de temps i diners.

Aquest projecte tracta de resoldre un problema de planificació de rutes. Aquest problema es basa en les nocions primàries d'un TSP (Problema del viatger). Tracta de resoldre un problema en el qual ens trobem en una ciutat en un e-scooter i volem visitar una sèrie de llocs d'aquesta ciutat en el menor temps possible.

Per arribar a tots els llocs que són un objectiu per a tu, cal tenir en compte la bateria de l'e-scooter, ja que va disminuint amb la distància recorreguda. Així que cal considerar si val la pena desviar-se de la ruta més ràpida per agafar un altre e-scooter que tingui prou bateria per arribar a la següent destinació, el canvi d'e-scooter afegeix temps extra (temps en el qual es canvia d'un e -scooter a un altre).

L'objectiu de la tesi és dissenyar i implementar un algoritme que resolgui el problema, és a dir, que li mostri la ruta més ràpida i el temps que triga a visitar totes les destinacions en el menor temps possible.

Resumen

Según la Wikipedia; El problema de enrutamiento de vehículos (VRP) es un problema combinatorio de programación entera y optimización que se pregunta "¿Cuál es el conjunto óptimo de rutas que debe recorrer una flota de vehículos para entregar a un conjunto dado de clientes?".

Han pasado muchos años desde que Dantzig y Ramser introdujeron este problema en 1959. Describieron una aplicación del mundo real relativa a la entrega de gasolina en estaciones de servicio y propusieron la primera formulación de programación matemática y un enfoque algorítmico. Pero aún así, la optimización de rutas es ahora más importante que nunca. Las grandes empresas de reparto invierten gran parte de su capital en la consultoría de VRP, ya que conocer la ruta más óptima para el reparto les ahorra mucho tiempo y dinero.

Este proyecto trata de resolver un problema de planificación de rutas. Este problema se basa en las nociones primarias de un TSP (Problema del viajante). Trata de resolver un problema en el que nos encontramos en una ciudad en un e-scooter y queremos visitar una serie de lugares de esa ciudad en el menor tiempo posible.

Para llegar a todos los lugares que son un objetivo para ti, hay que tener en cuenta la batería del e-scooter, ya que va disminuyendo con la distancia recorrida. Así que hay que considerar si merece la pena desviarse de la ruta más rápida para coger otro e-scooter que tenga suficiente batería para llegar al siguiente destino, el cambio de e-scooter añade tiempo extra (tiempo en el que se cambia de un e-scooter a otro).

El objetivo de la tesis es diseñar e implementar un algoritmo que resuelva el problema, es decir, que le muestre la ruta más rápida y el tiempo que tarda en visitar todos los destinos en el menor tiempo posible.

Dedication: *Li dedico aquest estudi a la meva familia, al meu company de pis, a la gent que he conegut durant el Erasmus, i a tota la gent que m'ha recolzat des de Barcelona*

Acknowledgment

I would like to thank my supervisor at CTU, Marek Cuchý, who, although there were times during the project when we could not understand each other, never let me down, helped me, and guided me through the project. I would also like to make a special mention to my supervisor at the UPC, Juan José Costa, who gave me very good advice in the hardest moments.

I would like to thank my parents, who have supported me at all times and have made me go ahead with the project when I was about to give up.

I would also like to thank my flatmate, Arnau, who has also supported me a lot and fixed my computer so that I could continue working on the project.

Finally, I would like to thank all the people I love in Barcelona. The fact that being far away from them has made me love them more than ever.

Revision history and approval record

Revision	Date	Purpose
0	06/05/2021	Document creation
1	16/06/2021	Document revision
2	21/06/2021	Document submission

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Marc Gracia i Riera	**marcg1998@gmail.com**
Marek Cuchý	**marek.cuchy@agents.fel.cvut.cz**
Juan José Costa Prats	**jcosta@ac.upc.edu**

Written by:		Reviewed by:	
Date	12/06/2021	Date	16/06/2021
Name	Marc Gracia i Riera	Name	Marek Cuchý Juan José Costa Prats
Position	Project Author	Position	Project Supervisor

Contents

List of Figures **11**

List of Tables **11**

1 Introduction **12**
1.1 Statement of purpose . . . 12
1.2 Requirements and specifications . . . 12
1.3 Methods and procedures . . . 13
1.4 Work Plan . . . 13
1.4.1 Work Breakdown Structure . . . 14
1.4.2 Milestones . . . 14
1.4.3 Gantt Diagram . . . 15
1.5 Deviations of the original plan and and incidences . . . 15

2 State of the art of the technology **16**
2.1 Some Data Structures . . . 16
2.1.1 Linked List . . . 16
2.1.2 Stack . . . 16
2.1.3 Queue . . . 17
2.1.4 Graph . . . 18
2.2 Search Algorithms . . . 19
2.2.1 Dijkstra's Algorithm . . . 19
2.2.2 A* Algorithm . . . 20
2.2.3 Dijkstra vs. A* . . . 20
2.2.4 Travelling salesman problem . . . 21
2.3 Heuristics . . . 22
2.3.1 Euclidian distance . . . 23
2.3.2 Manhattan distance . . . 24
2.3.3 Haversine distance . . . 25

3 Methodology / project development **27**
3.1 Construction of the problem . . . 27
3.1.1 Downloading the map . . . 27
3.1.2 How the input data is loaded . . . 28
3.2 Solution approach . . . 28
3.2.1 Description of the algorithm . . . 29
3.2.2 Important functions of the algorithm . . . 31
3.2.3 Pseudocode . . . 33

4 Implementation **35**
4.1 Vertex Class . . . 35
4.2 State Class . . . 36
4.3 Testing . . . 36

5 Experiments and results **37**

5.1 Experiments 37
5.1.1 Solution example 40
5.2 Results 40
5.2.1 Commentaries about the charts 41
5.2.2 Charts 43

6 Budget **48**

7 Conclusions **49**

8 Future Work **50**

References **51**

List of Figures

1 Work Breakdown Structure . . . 14
2 Gantt Diagram . . . 15
3 Linked List Diagram . . . 16
4 Stack Diagram . . . 17
5 Queue Diagram . . . 18
6 Graph Diagram . . . 18
7 example of TSP solution . . . 22
8 The distance between two points P and Q in the plane satisfies the Pythagorean theorem. 23
9 Eixample de Barcelona . . . 24
10 In Euclidean geometry, the green line has length $6\times\sqrt{2}\approx 8.48$, and is the only shortest path. In Manhattan geometry, the other lines have length 12, so it is not shorter than the other paths. . . . 25
11 Representation of "spherical triangles". . . . 26
12 Visualisation of a graph in QGIS . . . 28
13 Graph for the experiments . . . 37
14 Map with the goals to visit (blue dots) . . . 38
15 Example of a route solution for a random input data . . . 40
16 1 Goal . . . 43
17 2 Goals . . . 44
18 3 Goals . . . 44
19 150 e-scooters . . . 45
20 760 e-scooters . . . 45
21 1520 e-scooters . . . 46
22 100 battery . . . 46
23 15 battery . . . 47
24 Random battery . . . 47

List of Tables

1 Milestones . . . 14
2 Real e-scooters scenario . . . 38
3 Time in microseconds of the experiments to look at the behaviour of the A* and Dijkstra algorithms according to the number of goals to be visited. 41
4 Time in microseconds of the experiments to look at the behaviour of the A* and Dijkstra algorithms according to the number of e-scooters. . . . 42
5 Time in microseconds of the experiments to look at the behaviour of the A* and Dijkstra algorithms according to the battery of the e-scooters. . . . 42
6 Budget . . . 48

1 Introduction

1.1 Statement of purpose

The goal of the thesis is to design and implement an algorithm solving an e-scooter route planning problem, and to make an evaluation of the algorithm to see how it behaves in different scenarios.

What motivated me to do this project is that more and more we need to get to places quickly. We waste a lot of our time commuting between home and work for example.

I wanted to find a problem that was based on route optimisation for vehicles. Especially electric vehicles, as I consider them to be the future.

In this way, I could learn new fields that I consider to be of vital importance. Large delivery companies are constantly looking to optimise the delivery route of their vehicles, as it saves time and money.

I have chosen to make this route planning problem about e-scooters for the following reason. This reason is that it is a transport vehicle that anyone can use as no licence is needed.

1.2 Requirements and specifications

- **Requirements:**
  If I wanted to solve a route planning problem. First of all, I had to have a route planning problem. At least an idea of what route planning problem I could solve. So, I together with Marek Cuchý came up with the idea to come up with the following problem to solve.

  Basic structure of the problem:

  1. **Map of the city.** We have a map to work on.
  2. **Location of e-scooters.** Across the map, we have e-scooters randomly placed at different points on the map.
  3. **List of places.** The starting point and the goals where the user wants to go are located on top of the same map.

  Once we have the environment set up, we want to find the fastest route to all the goals the user wants to visit. To do this, we have to take into account the distance from the origin to the sites to be visited and the battery of the e-scooter

- **Specifications:**
  Now we have an idea of what problem to solve. For this project, what came next was to construct the problem and solve it.

We could separate the type of search into two parts, one for the construction of the problem and one for the solution itself. We would have to do an intensive study on data structures, specifically in Java for the construction of the problem. Then another intensive study on search algorithms for the solution of this problem. In addition, when it came to doing the research on these algorithms, it forced me to carry out other types of search that you will see later on, such as heuristic distances.

1.3 Methods and procedures

This project is made from scratch, it is not the continuation of any project. This project was proposed by Marek Cuchý from CTU to the student Marc Gracia i Riera from ET-SETB degree of UPC and accepted by Juan José Costa Prats from UPC.

From CTU, through Marek Cuchý, a Python script was provided to Marc Gracia i Riera. This Python script allows downloading OSM[1] data from a given region by entering its coordinates, processing them and converting them into a graph. The graph is saved in geojson files. The Python script is executed in the text editor Sublime Text 3.

The rest of the project is done in Java language in NetBeans IDE by Marc Gracia i Riera.

1.4 Work Plan

The methodology to organize the project is the same used at the rest of the degree. The project is splitted in a work breakdown structure with different work packages. A Gantt diagram is elaborated taking in account that changes on time plan may depend on circumstances such as machines maintenance, broken samples or other inconveniences that we cannot control.

[1] https://learnosm.org/en/osm-data/data-overview/

1.4.1 Work Breakdown Structure

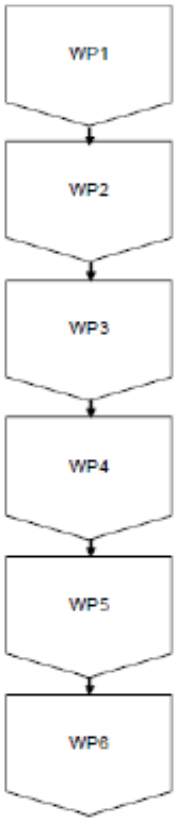

Figure 1: Work Breakdown Structure

1.4.2 Milestones

WP	Short title	Milestone/deliverable	Date (week)
1	Idea Problem	DOC about project idea	08/02/2021
2	OSM Data and Script	Script and map	22/02/2021
3	Studies	-	28/02/2021
4	Data Structure/Algorithm	Netbeans Project	05/05/2021
5	Idea experiments	DOC about experiments idea	20/05/2021
6	Testing experiments and conclusions	Charts and conclusions DOC	10/06/2021

Table 1: Milestones

1.4.3 Gantt Diagram

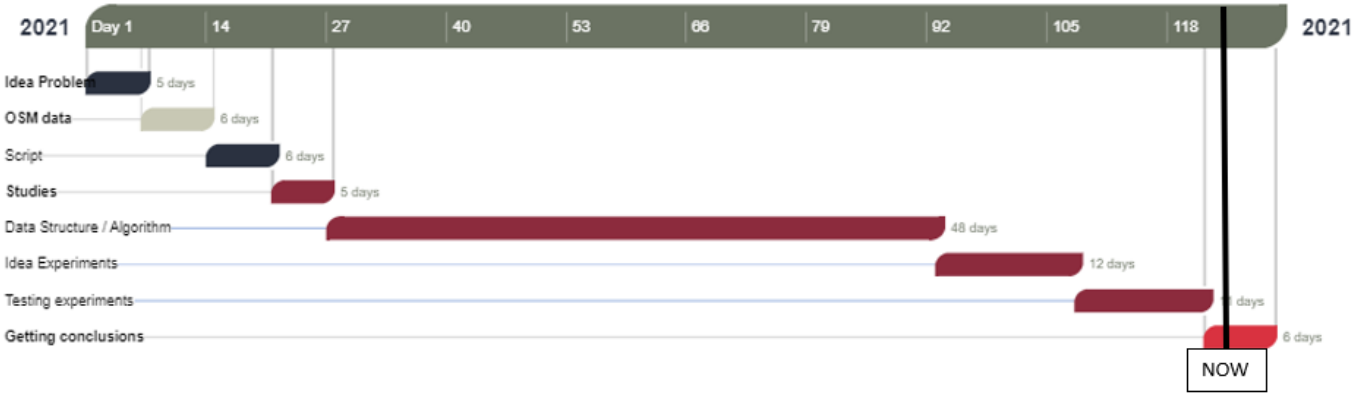

Figure 2: Gantt Diagram

1.5 Deviations of the original plan and and incidences

The project started in mid-February 2021. This year has been particularly complicated due to the SARS Covid-19 pandemic. The limitations of mobility, prevention measures and other things derived from it, made us adapt our timetable to the government's indications. However, we were aware of this problem from the beginning, so adapting the work plan was not a big problem.

The project started very well, I knew what to do each week and was keeping to the timetable as planned. A month and a bit into the project, I started to get stuck. I didn't know what algorithm; Dijkstra, A*, ... my algorithm would be based on, every week I was changing algorithm and data structure. It felt like every week I was starting the project all over again.

Once I had decided on which algorithm I was going to base it on, and I had the data structure well thought out. The problems started when it came to knowing how to implement it. All these problems caused me to fall behind with the project.

However, I overcame all these problems and managed to get the project on time.

2 State of the art of the technology

In this section we will explain three theoretical aspects necessary for the realisation of the project. We will talk about basic data structures, search algorithms and heuristics needed to solve our project.

2.1 Some Data Structures

Data structures are a way of organising data in the computer in such a way as to allow us to perform operations with it in a very efficient way. Depending on the algorithm we want to execute, there will be times when it is better to use one data structure or another structure that allows us more speed.

Therefore, it is very important to know what kind of data structures we can have. In this way, make a study of them to know which data structure suits you for your algorithm. Much of the information below is taken from Book [1].

2.1.1 Linked List

The Simple Linked List is the most fundamental pointer-based data structure, and the other data structures are derived from its fundamental concept.

The linked list allocates space for each separate element in its own block of memory, called a node. The list connects these nodes using pointers, forming a string-like structure.

A node is an object like any other, and its attributes will do the work of storing and pointing to another node. Each node has two attributes: a "content" attribute, used to store an object; and a "next" attribute, used to refer to the next node in the list.

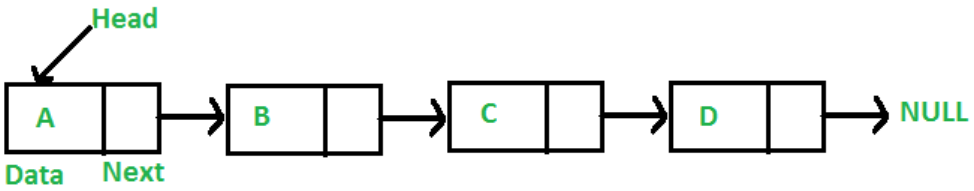

Figure 3: Linked List Diagram

2.1.2 Stack

The stack is a structure based on the LIFO concept, i.e. the last element in is the first element out. Stack is a very simple data structure.

Imagine you have a toy gun that shoots balls, and, to load it, you have to feed the balls one by one down the front of the gun barrel, one after the other. The first ball you shoot will be the last one you put in, and the last one you shoot will be the first one you put in. That's a stack.

The implementation of a stack is very similar to that of a linked list, and only differs in the way we manage the stored elements. In a stack, we will create methods that fulfil the functions outlined below and clarified with Fig. 4, i.e. one method that adds a node to the top of the stack and one method that removes the first node from the stack. For this implementation, the nodes will be instances of the Node class, defined in the same way as we defined it for the linked list.

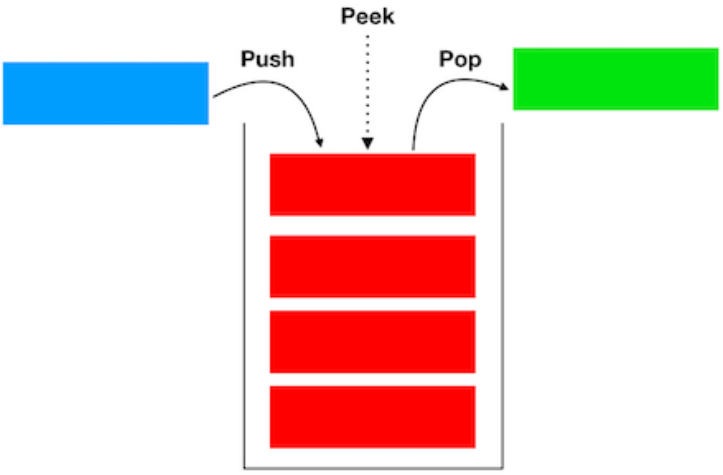

Figure 4: Stack Diagram

2.1.3 Queue

The queue is a structure based on the FIFO concept, i.e. the first item in is the first item out.

Like stacks, the implementation of the queue is very similar to that of the linked list, and differs only in the way we manage the stored elements. In a queue, we will create methods that fulfil the functions outlined below and clarified with Fig. 5, i.e. a method that adds a node to the back of the queue and a method that removes the first node from the queue.

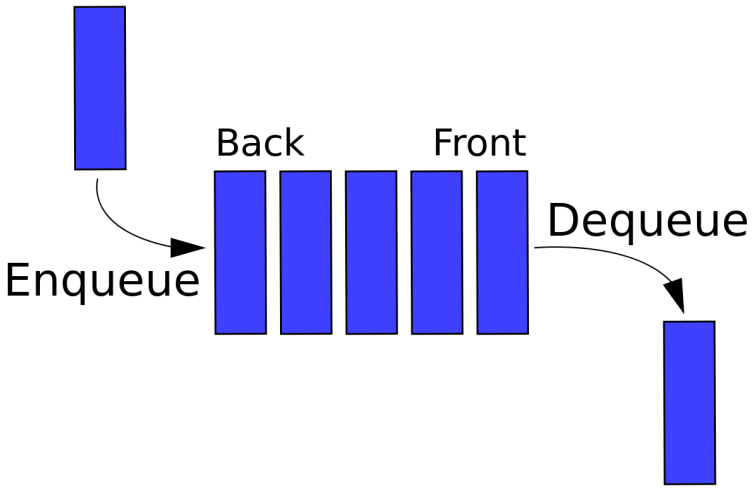

Figure 5: Queue Diagram

2.1.4 Graph

From the concepts explained above, with special mention of the linked list concept, we can construct a graph.

A graph is a non-empty set of objects called vertices (or nodes) and a selection of pairs of vertices, called edges, which can be oriented or not. Typically, a graph is represented by a series of points (the vertices) connected by lines (the edges). In addition, edges can have a weight.

There are many types of graphs, but we won't go into too much depth here; this article [2] explains graph theory in more detail.

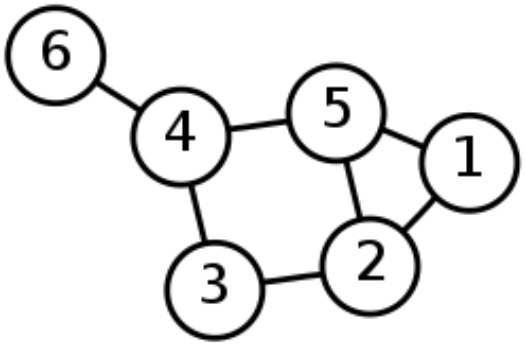

Figure 6: Graph Diagram

If you are interested in learning more about data structures in Java, I recommend this article [3].

2.2 Search Algorithms

A search algorithm can have many functionalities depending on the context. In our context, we will use it to find a solution in a data structure.

In this section we will focus on two search algorithms, Dijkstra and A*, which have been studied for this project. Later, we will explain the travelling salesman problem, a problem that inspired us to solve ours.

2.2.1 Dijkstra's Algorithm

Dijkstra's algorithm [4]. Also called the minimum paths algorithm, it is an algorithm for determining the shortest path given a source vertex to the rest of the vertices in a graph with weights on each edge. Its name refers to Edsger Dijkstra, who first described it in 1959.

- It is a greedy algorithm.
- It works in stages, and takes the best solution at each stage without considering future consequences.

The Dijkstra algorithm is solved as follows:
Given a weighted directed graph of N non-isolated nodes and with non-negative weights, let x be the initial node, a vector D of size N will store at the end of the algorithm the distances from x to the rest of the nodes.

1. Initialise all the distances in D with an infinite relative value since they are unknown at the beginning, except that of x which must be set to 0 since the distance from x to x would be 0.
2. Let $a = x$ (we take a as the current node).
3. We traverse all adjacent nodes of a, except for the marked nodes, we will call these unmarked nodes V_i.
4. For the current node, we calculate the tentative distance from that node to its neighbours with the following formula: $d_t(V_i) = D_a + d(a, V_i)$. That is, the tentative distance of node V_i is the distance that node currently has in vector D plus the distance from node a (the current node) to node V_i. If the tentative distance is less than the distance stored in the vector, we update the vector with this tentative distance. That is: If $d_t(V_i) < D_{Vi} \rightarrow D_{Vi} = d_t(V_i)$.
5. We mark node a as complete.
6. We take as the next current node the one with the smallest value in D (this can be done by storing the values in a priority queue) and go back to step 3 as long as there are unmarked nodes.

Once the algorithm is finished, D will be completely full.

In this paper [5], an interesting application of the Dijkstra algorithm for the planned route of a robot is carried out.

2.2.2 A* Algorithm

A* [6] is an intelligent or informed search algorithm that searches for the shortest path from an initial state to the goal state through a data structure, using admissible heuristics.

An admissible heuristic is the one that guarantees the finding of an optimal path between the start node and the goal node, if such a path exists. In the search A* an admissible heuristic is one that does not overestimate the remaining distance between the current node and the target node.

A* is an informed algorithm that bases its behaviour on the evaluation of a function expressed as follows:

- $g(n) \equiv$ the cost of the distance made from the origin node to the current node
- $h(n) \equiv$ the heuristic function. It represents the estimated cost of the best path from the current node to the goal node.
- $f(n) = g(n) + h(n) \equiv$ this sum gives an approximate distance from the origin to the target.

In pathfinding, the heuristic function is usually the straight path to the goal, since no matter what the map is like.

In addition, I extracted a lot of important information through paper [7], like the concept of open and closed list for nodes. This paper gives us a visualisation of A* search at the multi-objective level.

2.2.3 Dijkstra vs. A*

A* is generalization of Dijkstra, the only difference is that A* tries to search for a best path by using a heuristic function that gives priority to nodes that are supposed to be better than others, while Dijkstra simply explores more sub paths. This is because in the function $f(n) = g(n) + h(n)$, $h(n) = 0$ in Dijkstra

The optimum depends on the heuristic function used, can may return a sub-optimal result because of this and at the same time, the better the heuristic for your specific design, the better the result (and possibly the speed). If the heuristic is admissible, A* finds an optimal solution

It is bound to be faster than Dijkstra even if it requires more memory and more operations per node, as it scans far fewer nodes and the gain is good in any case, but not always happen. So, we have to consider that may we can have a consistent heuristic that however takes a lot of time to calculate which can make Dijkstra's algorithm faster in practice.

This paper [8] makes a real comparison between the Dijkstra algorithm and A*, which I find very interesting. It makes you see from another perspective their utilities when it comes to search.

2.2.4 Travelling salesman problem

The following information is extracted from the following paper [9].

- TSP's description: We have a number of nodes (cities, towns, localities, shops, companies, etc.) connected by edges with a weight that must be visited by an entity (person, travelling agent, car, plane, bus, etc.), without visiting the same node twice. If we have 3 nodes (a, b and c) to visit, then we would have a function of permutations c(3,2), that is, we would have 6 possible solutions: abc, acb, bac, bca, cab, cba, for the case of 4 nodes we would have 12 combinations, for 10 nodes we would have 90 combinations, for 100 cities we would have 9,900 combinations and so on. As an example in the problem of Homer's Ulysses who tries to visit the cities described in the Odyssey exactly once (16 cities) where there are multiple connections between the different cities, Grötschel and Padberg (1993) came to the conclusion that there are 653,837'184,000 different routes for the solution of this problem.
- Basic Algorithm: The TSP is considered as a set of graphs whose edges are the possible paths that the entity can follow to visit all the nodes, and whose algorithm can be represented as follows:

Algorithm 1 Algorithm based on a Travelling Salesman Problem

procedure (INPUT)$C(i,j) \rightarrow i,j = 1..N$ ▷ Number of cities N and array of costs (weights between the nodes), we begin from city number 1
  Starting values
  $C \leftarrow 0$
  $cost \leftarrow 0$
  $visits \leftarrow 0$
  $e \leftarrow 1$ ▷ e = pointer of the visited city
  for $r = 1$ to $N - 1$ **do**
    Choose of pointer j with
    $minimum = C(e,j) = min_C(e,k); visits(k) = 0$ and $k = 1..N$
    $cost = cost + minimum$
    $e = j$
    $C(r) = j$
  $C(n) \leftarrow 1$
  $cost = cost + C(e,1)$
  OUTPUT ▷ Vector of cities and total cost

- Characteristics: The TSP is classified as a Combinatorial Optimisation Problem, i.e. it is a problem involving a certain number of variables where each variable can have N different values and whose number of combinations is exponential, which gives rise to multiple optimal solutions in theory (solutions that are calculated in a finite time) for an instance.

  TSP is a problem considered difficult to solve, being called in computational language NP-Complete, that is, it is a problem for which we cannot guarantee that the

optimal solution will be found in a reasonable computational time. Different methods are used to provide a solution, among which the main ones are called meta-heuristics whose objective is to generate good quality solutions in much shorter computation times (time-response optimal solutions).

As we can see, this problem bears some resemblance to our own. Like us, in this problem you are given certain places to reach in an optimal way, getting to all the places as quickly as possible.

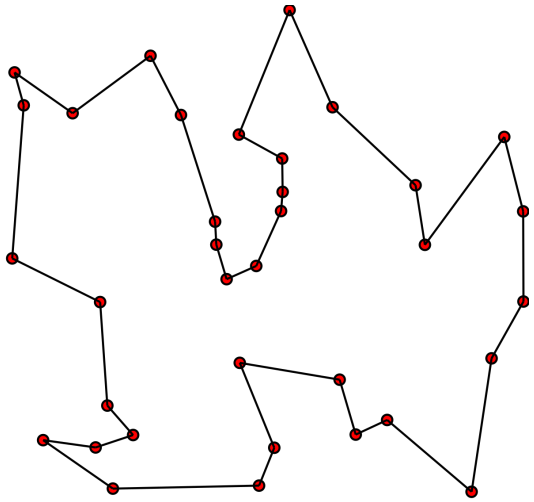

Figure 7: example of TSP solution

2.3 Heuristics

When I discovered the A* algorithm. I understood that the key to a good performance of its algorithm was to find a good method of calculating heuristic distances between two points. So I had to do an intensive research for heuristic distance calculations in case I finally chose to base my algorithm on the A* algorithm.

- What is a heuristic distance?
  Heuristics are criteria, methods or principles for deciding which of a number of actions promises for deciding which of several actions promises to be the best to achieve a given goal.
  The use of heuristics allows us to guide our search for a solution. Which will allow us to us to obtain a solution more quickly than if we blind search strategies.
  In search problems:
  - A heuristic heuristic is be a function that we will use to estimate how close we are to the goal.
  - Each heuristic will be designed for a particular search problem. Hence, we first have to do a study to see what type of heuristic can best suit our project.

After a long research. I found three methods for calculating heuristic distances between two points. In my case, these two points would be two nodes within my graph. These three methods are: euclidian distance, Manhattan distance and haversine distance.

2.3.1 Euclidian distance

The Euclidean distance is a positive number that indicates the separation of two points in a space where the axioms and theorems of Euclidean geometry are satisfied.

The distance between two points A and B in a Euclidean space is the length of the vector AB belonging to the only straight line passing through these points.

Two-dimensional Euclidean space is a plane. The points of a Euclidean plane satisfy the axioms of Euclidean geometry, for example:

- A single straight line passes through two points.
- Three points on the plane form a triangle whose internal angles always add up to 180°.
- In a right triangle the square of the hypotenuse is equal to the sum of the squares of its legs.

In two dimensions a point has X and Y coordinates.

For example a point P has coordinates (x_P, y_P) and a point Q coordinates (x_Q, y_Q). The Euclidean distance between point P and Q is defined by the following formula:

$$d(P,Q) = \sqrt{(X_q - X_p)^2 + (Y_q - Y_p)^2}$$

It should be noted that this formula is equivalent to the Pythagorean theorem, as shown in Figure 8.

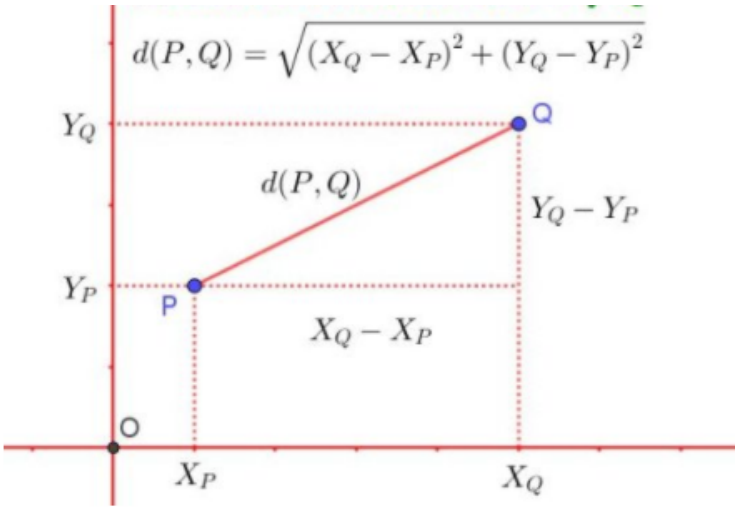

Figure 8: The distance between two points P and Q in the plane satisfies the Pythagorean theorem.

2.3.2 Manhattan distance

The Manhattan distance tells us that the distance between two points is the sum of the absolute differences of their coordinates. That is, it is the sum of the lengths of the two legs of the right triangle. Something like the length of any staircase going up from A to the point B. A route linking point A and B through horizontal and vertical segments.

The Manhattan distance is based on the calculation of the Euclidean distance plus the possibility of avoiding obstacles. It is always a longer distance, but it is closer to a real distance. As long as, between those two points in real life you can encounter obstacles if you go in a straight line.

A clear example is Barcelona's Eixample, Fig.9. If you go in a straight line between two points that do not have a slope of 0 with respect to the coordinate axes you will crash into the buildings. On the other hand, if you apply the Manhattan distance, the distance is longer but you avoid crashing into the buildings.



Figure 9: Eixample de Barcelona

It can be perfectly understood thanks to figure 10. Where the difference between the two heuristics is shown graphically.

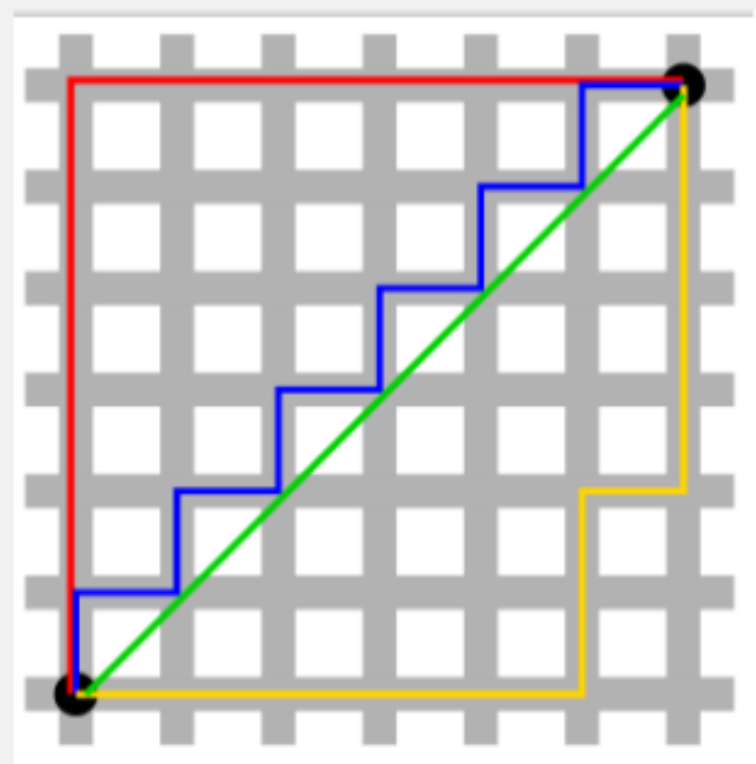

Figure 10: In Euclidean geometry, the green line has length $6 \times \sqrt{2} \approx 8.48$, and is the only shortest path. In Manhattan geometry, the other lines have length 12, so it is not shorter than the other paths.

2.3.3 Haversine distance

The haversine formula is an important equation for astronomical navigation, in terms of calculating the great-circle distance between two points on a globe by knowing their longitude and latitude. It is a special case of a more general formula of spherical trigonometry, the haversine law, which relates the sides and angles of "spherical triangles".

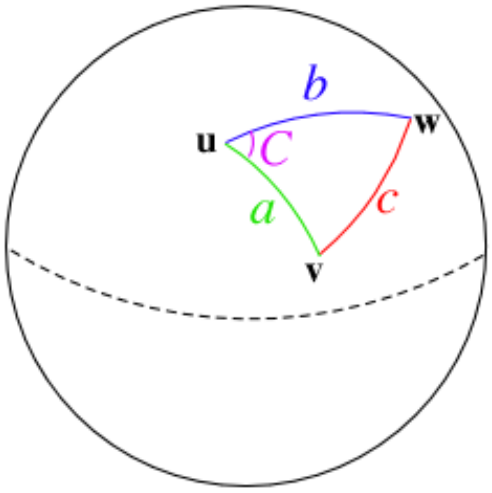

Figure 11: Representation of "spherical triangles".

For the calculation of any pair of points on a sphere. We can use the following formula.

$$haversin(\frac{d}{D}) = haversin(\varphi_1 - \varphi_2) + cos(\varphi_1)cos(\varphi_2)haversin(\triangle\lambda)$$

Where:

- $haversin \equiv$ haversine's function. $haversin(\theta) = sin^2(\frac{\theta}{2}) = \frac{(1-cos(\theta))}{2}$
- $d \equiv$ the distance between two points.
- $R \equiv$ the radius of the sphere.
- $\varphi_1 \equiv$ the latitude of point 1.
- $\varphi_1 \equiv$ the latitude of point 2.
- $\triangle\lambda \equiv$ the length difference.

Then, to find the distance d, it can be done simply as follows:

$$d = R * haversin^{-1}(h) = 2R * haversin(\sqrt{h})$$

Where:

- h is $haversin(d/R)$

The following article [10] is interesting, as it explains heuristic distances in transport applications.

3 Methodology / project development

This project, as mentioned above, is separated into two parts; the construction of the problem and its solution.

After having done the intensive study for the two parts, it was time to get down to work. We started by constructing the problem and then solving it.

3.1 Construction of the problem

The construction of the problem has two clear parts, the downloading of the map and the uploading of input data to the map. We will explain these two parts step by step at a high level. But it is worth mentioning that after intense study, we decided that the best data structure we could have was a graph, with its vertices and edges with a weight, this weight being the actual distance between the nodes.

3.1.1 Downloading the map

As mentioned above, I was given a script from the CTU. This script is written in Python language.

The script allows you to download an area of the map from OSM. You enter the coordinates of the map area you are interested in. Then, in a directory, two files in geojson format[2] are created for you. These files are the nodes and another one the edges that join the nodes, allows you to build a graph.

In order to visualise a graphical representation of the graph we have just downloaded. We use the software QGIS[3]; Quantum GIS is a programme for visualisation, editing and analysis of data that makes up a geographic information system.

Thanks to QGIS we can visualise the geojson files that the script has just given us on an OSM layer. In this way, we can see perfectly the distribution of the nodes of the graph on the map. In addition, we can see the weight of the edges, which is the real distance between nodes in centimetres. Also, it gives us more information, but it is not necessary for our project.

In the picture below. We can see a small graph obtained from the Python script and visualised from QGIS. You can see labels indicating the indexes of the nodes, and also, the weight of the edges, as we said before, in centimetres.

[2]https://en.wikipedia.org/wiki/GeoJSON
[3]https://www.qgis.org/en/site/



Figure 12: Visualisation of a graph in QGIS

3.1.2 How the input data is loaded

Once we have the geojson files, we have to load them into our programme in order to be able to work with them.

To do this, as we said before, we will work with the Java language on Netbeans IDE.

To load these files, we will use the Jackson Java library[4]. Jackson is a java library that allows us to convert classes to JSON text and vice versa. To do this we have to build classes that have the same names as the attributes in the geojson files. In this way, we can make the data in the files readable.

We use the Jackson library in a class called Graph. This class takes the data from the files and returns a list of vertices.

We now have a graph built that allows us to start working on it. So all we need to do is to add the e-scooters and the sites to be visited to the graph.

To do this, we will rely on a class called Vertex, to learn more about the Vertex class, see the section 4.1. If we want a node to have an e-scooter or a place to visit, we take the Vertex object that refers to that node. For it to have an e-scooter, we set a battery value to that e-scooter and for it to be a place to visit, with a boolean we set that node to be a place to visit.

3.2 Solution approach

We will now explain the main part of this project. The search algorithm. We are going to explain how it is built, on which search algorithm it is based and its main functions.

First of all. We need to know what data we are passing to the algorithm. We pass two pieces of data to the main function in charge of running the whole algorithm. A vertex

[4]https://www.tutorialspoint.com/jackson/index.htm

list, which is a list of all the loaded information that is not created in the algorithm. That is, the vertex list comes with all the input data already loaded. In addition, the other parameter that is passed to it is the source node.

For the construction of the algorithm, we have based ourselves on the A* search algorithm but with multi-objectives, dominance and pareto-sets. Recall the basic operation of the A* algorithm in section 2.2.2. Since we will use heuristic distances to find in a faster way the target nodes. We will also apply Dijkstra, we just have to change the heuristic distance to 0.

We will now focus on the description of the algorithm. We will go over how it works with the support of a pseudocode. Finally, we will go in depth into three aspects of the algorithm; how the time/distance is calculated, how the battery is calculated and the concept of dominance.

3.2.1 Description of the algorithm

As mentioned above, our algorithm is based on the A* model. The basis of our algorithm is a priority queue of states.

But what are states? States are objects that are created on vertices, and have 4 main attributes; the index of the vertex, the actual time it takes from the origin to the current node plus the heuristics and the current battery that the e-scooter has and the locations that have been visited.

The states are stored in the priority queue prioritising the state with the least amount of time to reach the next goal. Then, if the time between two states is the same, they are ordered according to the battery prioritising the one with the highest battery is chosen first. For a better understanding of states, I explain the State class in section 4.2. This section goes into more detail about states, and their attributes.

We will now start by describing the algorithm without going into detail, but at a high level. In section 3.2.3, there is the pseudocode of the algorithm that will help you to follow the explanation.

Once we receive the graph and the source node, we can start. It is important to know that in the graph, the places to visit are those that have the boolean *Place* set to true, and that the e-scooters are located in those nodes where their battery is greater than 0.

To start solving the problem, the first thing to do is to arrange the order in which we want to visit the goals entered by the user in a simple way, by the nearest neighbor. To find out which locations are closer, we do this by calculating a heuristic distance between nodes. After all the study done on the calculation of heuristic distances, we decided to use **Haversine** because we understood that it was an admissible heuristic to be able to solve our problem, see in section 2.3.3.

Now, starting from the origin vertex, we take out the origin state. To create a state, we have to pass it all the necessary attributes of the vertex. Once we have this state, we add it to the priority queue and to the open state list of the source vertex.

The open list is for those states that we have not yet created new states of their neighbours from this one. See section 3.2.2 in "dominance section" for a better understanding of how we sort this list.

We will now enter the loop. We will not exit this loop until the priority queue is completely empty or found a solution.

Inside the loop, the first thing we have to do is to take the first state out of the queue. From this state we pull out the next goal that has to reach this state, this depends directly on the number of targets it has currently visited. What it does is that if for example this state has already visited a goal, we will set the next goal to visit to be the one that is in the number one position in the list of targets. Also, from this state we get the Vertex object that has the same index as the state.

Once this is done, we take the state we have taken from the open list, this state is the same as the one we are working with that we have extracted from the priority queue, of this vertex and add it to the closed list of states. In this way we tell the algorithm that we will not see any more of this state once we have looked at all its connections to its nearby nodes.

Once we have the vertex, we look to see if it is the node we want to reach. If it is, we put that the state we are working with has reached one more target. If by adding one more target, this state has reached all targets, we add it to a new priority queue, although it could be a list. It could be a list and always extract the first state from the list, since the first state we get is always the optimal solution. This new priority queue is ordered in the same way as the previous one, but only states that have reached all targets are added. But if this is not the case, what we do is set the new target that this state has to reach.

Once we have looked at all this, we will go into the expand function. This function is where the new states are created.

We enter a new loop. This loop is used to look at all the neighbour nodes of the node where we are currently located.

The first thing we do is to see if we can reach the neighbouring node we are targeting with the battery we have. If we can't reach it, then we stop targeting that neighbouring node and look at the next neighbouring node. On the other hand, if we can reach it, what we do is to continue with the process of creating new states.

If the neighbouring node has no e-scooter, we will only create a single new state. The important thing for the creation of an e-scooter is to create it with the index of the vertex in which it is located, pass it the times, the battery that the e-scooter has once it has reached that neighbouring node and finally tell it how many targets it has already visited. This information is taken from the state with which we are working and we tell it that the new state is created from this one.

Things change a bit if this neighbouring node has an e-scooter. This will create two states instead of one. One will be created in the same way as before. Then, we will

create another one, in which we will represent that we are changing e-scooter. This new state will collect all the information of the state from which it comes, but with modifications in time and battery. The battery of the new state will be the battery of this e-scooter that we have found, and in the time we will have a penalty. This penalty is 5 minutes, which represents the time to change from one e-scooter to another.

Once we have created the states we have to look at whether or not they are dominated by other states. If this new state is dominated by any other state in the open or closed list of states, this new state is not retained. Since there are states that are better than this new state.

´Once we have checked that this new state is not dominated, we do two things; add this state to the priority queue and to the open state list and see if this state dominates another state in the open state list. If it does, what we do is remove the dominated state from the open list and from the priority queue.

Once we have done this, we look at the next neighbouring node and do the same process again. On the other hand, if we have already looked at all the neighbouring nodes, we go back to the top. We take the first state in the queue and go through the whole process again.

When the queue is empty. It will mean that we have already looked at all possible paths to reach all targets. What we do then is to take the first state from the priority queue of the states that have managed to reach all the targets. From this state we go through the whole path of nodes that we have done, even saying if we have changed e-scooter in any node, until the node from which we have started. We know this thanks to the fact that when we created a new state we told it from which state it came from. In addition, we can get information such as the total distance travelled and how long it took us to reach all the nodes.

To better understand how the battery is calculated, the time and how we look at whether one state dominates another, go to section 3.2.2.

3.2.2 Important functions of the algorithm

This section is created to better understand how the algorithm works. We go into detail on 3 important functions, the battery calculation, the time calculation and how to see if one state dominates another.

- **To calculate the battery:** To calculate the battery we use a function that is passed two parameters; the state we are working with ($actualState$) and the real distance in metres between the node I am located at and the neighbouring node I am targeting ($weight$).
  We set two established parameters; the maximum battery that the e-scooter can have ($maxBattery = 100$) and the maximum range in metres that it could have ($maxRange = 15000$). The range is chosen after looking at the different ranges of e-scooters on the market. In addition, we reduce the autonomy that the

manufacturer claims to have due to the calculation of the battery cycles already made. That is to say, they are not new e-scooters from the factory so that they are closer to a real case.
This function has two more parameters; the actual battery that the e-scooter I am riding has ($actualState_{battery}$) and the battery that I will have once I have reached the neighbouring node ($battery$). The latter is the one we return. To calculate it, we use a linear function that only depends on the distance travelled between the two nodes. We do not depend on the weight of the user, the surface of the road or whether or not there is a slope.

$$battery = actualState_{battery} - (weight * \frac{maxBattery}{maxRange})$$

It should be noted that if the battery calculation is negative, 0 is returned, i.e. it does not reach the neighbouring node.

- **To calculate the time/distance:** The fact that we have relied on A* for the construction of our algorithm is mainly due to this part. What we will do is to use the function $f = g + h$ for the calculation of the distance. If here, we set the value of h to 0, we switch to the Dijkstra algorithm. Remember the difference between Dijkstra and A* in section 2.2.3.

  As you already know from section 2.2.2. This function is used to measure an approximate distance from the origin to the next target. Where g is the actual distance from the origin to the node I am at and h is the heuristic (Haversine) distance from the node I am at to the next target.

  Then, to calculate the time we use the uniform rectilinear motion function. This function is $t = \frac{x}{v}$ where $t \equiv time$, $x \equiv distance$ and $v \equiv speed$. We have to set a default value for the speed, an average speed. In our case we decided to set $500\frac{m}{min}$.

- **Dominance:** It should be clear that if you want to look at whether one state dominates another, the indices have to match. Then you look at the other attributes, time, battery and number of targets achieved. It should also be noted that there are many states that do not dominate each other.

  Let's put it this way, one state dominates another state' if:
  - $index = index'$
  - $time <= time'$
  - $battery >= battery'$
  - $goalsArrived >= goalsArrived'$

  These 4 conditions have to be met in order to decide whether one state dominates another. There is one very important detail, and that is that the time to look at is different depending on whether we are looking at dominance in the priority queue or in the open list of states. In the queue we look at the actual time from the origin to the node plus the heuristic time from the node to the next target. Then, in the open state list we look only at the real time from the origin to the node we are at.

3.2.3 Pseudocode

Algorithm 2 Optimal Route Search

procedure (INPUT)$G(V, E) \rightarrow V = 0..N-1$ ▷ Each edge has the weight between vertices (weight real distance in meters)
    Starting values
    PQ ▷ priority queue of states (sorted by the time + heuristic)
    PQA▷ priority queue of states (sorted by the time + heuristic) already arrived to all goals
    $Goals \leftarrow$ V_i has $place = true$
    $Escooters \leftarrow$ V_i has $battery > 0$
    toSortGoals()
    $originVertex \rightarrow$ create new state: $actualState$
    add $actualState$ to PQ
    add $actualState$ to $OPENED_V$
    while PQ not empty **do**
        $actualState \leftarrow PQ$.poll()
        set $goalVertex$ depending on goals already visited
        get V from $actualState$
        remove $actualState$ from $OPENED_V$
        add $actualState$ to $CLOSED_V$
        if V==$goalVertex$ **then**
            $actualState$.set(number of places already visited + 1)
            if $actualState$ arrived to all goals **then** add $actualState$ to PQA
            set $newGoalVertex$ depending on goals already visited
        expand()
    OUTPUT Returns traceback from PQA.poll() ▷ Algorithm finishes

Algorithm 3 Expand function

```
1: procedure (INPUT) actualState
2:     for each targetVertex ∈ V do
3:         if calculateBattery()→ 0 then                ▷ doesn't arrive to targetVertex
4:             continue
5:         if targetVertex.battery == 0 then
6:             create newState
7:         else
8:             create two newState →one with the battery calculated and other with the
   battery from targetVertex + 5 minutes of extra time for switching e-scooters
9:         You have to set the predecessor of newState that is actualState and set the
   real distance already done in meters from source.
10:        if newState is dominated in CLOSED_targetVertex then
11:            continue
12:        if newState is dominated in OPENED_targetVertex then
13:            continue
14:        add newState to PQ
15:        add newState to OPENED_targetVertex
16:        for all state in OPENED_targetVertex do
17:            if newState dominates state then
18:                remove state from PQ
19:                remove state from OPENED_targetVertex
```

4 Implementation

In this section we take an in-depth look at the Vertex and State classes. We will also explain what we did to check that the algorithm satisfied what we wanted, i.e. delivered the optimal solution. This will help us to understand much better how this project is built. Above all, to better understand how the algorithm works.

4.1 Vertex Class

It is very important to understand what these vertices are. From these vertices all the information that the algorithm will work with is taken or stored.

Each vertex is a node extracted from the geojson file. The information extracted in the Graph class thanks to the Jackson library is stored in the Vertex class.

Understanding what this class is composed of, makes it possible to understand the project much better.

Each object of the Vertex class represents a node of the graph. The main attributes of this class are:

- **Node ID:** it is only necessary for the Graph class to return the list of vertices in order to build the graph. Since the geojson file that gives us the information about the edges, indicates the connections of the nodes from their ID's and not their indexes.
- **Node Index:** indicates the index of that node. It helps us to find the node in question easily. The range of the indices will always go from 0 to the value of the number of nodes in the graph minus 1. The ID, on the other hand, are larger numbers that identify the node according to its position in the world (latitude-longitude).
- **A list of adjacency list of objects of the Edge class:** which after a long study, we understand that it is a way to build a graph we felt comfortable with. The Edge class is simpler but very important as it links the nodes together. It returns the origin node, which is always the vertex that calls this list, the node to which it is connected and the weight of the edge in centimetres.

At the moment, all this information is collected in the Graph class. However, there are attributes that allow us to create different scenarios for our experiments with the algorithm; to draw routes, behaviours... For example:

- **Goal:** attribute that tells us if that node is a target to be visited by the algorithm.
- **Battery:** Then, there is an attribute that tells us if that vertex has an e-scooter there with a percentage of battery. That is, if this attribute is 0, we say that there is no e-scooter.

Finally, there are attributes that are used and loaded by the algorithm. In this case two state lists, one open and one closed.

4.2 State Class

The states are objects that trace the route to reach all the objectives within the graph, but not all of them manage to get there. We will explain their attributes, because this way we will have a good base when we explain the algorithm.

The state is collected by the State class. The state has 4 main attributes:

- **Index:** which is the same index on the node/vertex where it is located.
- **Time:** which would be the actual time it has been travelled plus a heuristic time remaining to reach the next goal.
- **Battery:** which is the battery that the e-scooter has at that moment.
- **Goals:** The number of targets it has already reached.

In addition, there are other attributes that are not the main ones. That is, they are not the attributes that actually define a state. They are also important for the development of the algorithm. These are attributes such as:

- **Real time:** the actual time that has elapsed not counting the changes from one e-scooter to another, we do not take into account a heuristic time.
- **Extra time:** this time allows us to add 5 minutes each time the e-scooter has been changed.
- **Predecessor state:** it is good to know from which state it has been created and then trace a route from the end point to the origin.
- **Real distance:** the actual distance travelled so far is also recorded.

4.3 Testing

Before I start to explain the results, I would like to mention how the algorithm was checked to ensure that it worked correctly. For this I had to do calculations by hand. I was checking if the battery was decreasing well as I went along, if I was getting the distance right. But of course, in graphs with hundreds or even thousands of nodes, it's not possible. So I was inspired to do it on graphs like the one in figure 12. As there were only a few nodes, I could follow the trajectory by hand. Of course, I was changing the percentages of the batteries, the number of targets, the distances between the nodes, I was changing the speed, etc. All these primary experiments gave me the confidence that the algorithm worked correctly and that it returned the optimal route.

5 Experiments and results

In this section we will explain the experiments we have carried out to look at the behaviour of the algorithm. We will look at the behaviour of the algorithm by making various changes to the number of targets, the number of e-scooters and even the battery percentage of the e-scooters.

5.1 Experiments

To start the experiments, we had to be clear about what data we had to enter. This input data could change according to two types; the targets and origin, and the e-scooters.

In addition, we had to have a graph on which to do the experiments. We decided on the one you can see in figure 13. It is a map of the city of Prague, this graph itself has more than 7000 nodes, 7602 to be more specific, its an area of $25km^2$. We understood that it was a map size on which we could draw good conclusions. It is even a map that favours us to have a real scenario. I mean, an e-scooter rental company could easily have e-scooters all over this area.

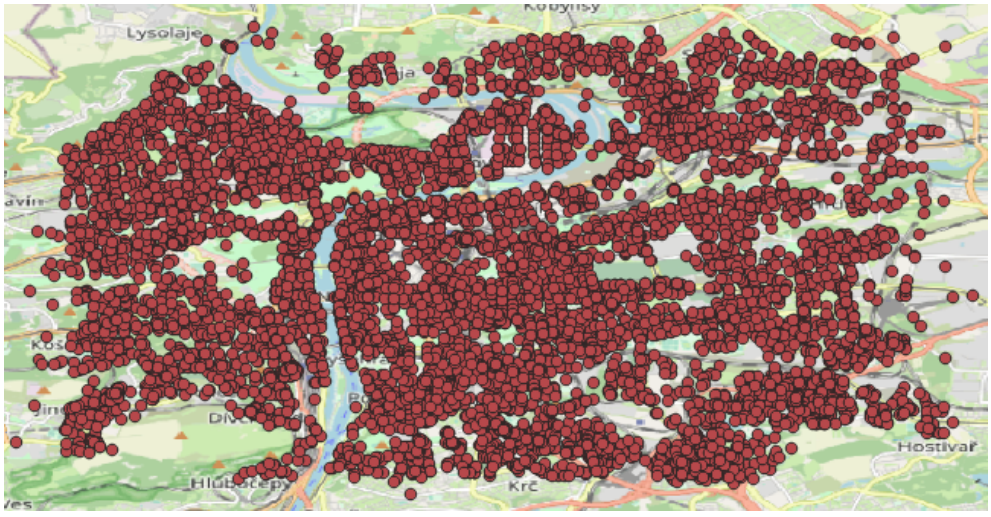

Figure 13: Graph for the experiments

- **Goals and source:** On goals, we have a fixed list of 20 goals (nodes) entered by hand (as shown in Fig.14). These nodes represent places of interest in the city of Prague.



Figure 14: Map with the goals to visit (blue dots)

So, what we do is we do 150 iterations. In each of these iterations we have a different origin and different goals. The number of goals varies depending on the experiment we are running, can be 1 goal, 2 goals or 3 goals. The goals are got randomly from the goals fixed list. The source is any node of the graph. For each of the 150 iterations, the battery percentage of the e-scooters is the same and they are located at the same nodes.

- **E-scooters:** For e-scooters we can modify two aspects. These two aspects are; the number of nodes an e-scooter has and the battery they have.
  Regarding the number of nodes that have an e-scooter, we divided it in three parts, 20 percent of the nodes have an e-scooter, 2 percent and finally 10 percent. This last one is the type that would be closest to a real case. We indicate this as a real case based on our own experience. We tried to find real data on how many e-scooters these rental companies have per square kilometre, but we did not get any results.
  Finally, we experimented with modifying the batteries. Also in three categories for the battery; all e-scooters have 100 percent of the battery, all have 15 percent and finally another category closer to a real case, also from our own experience when using apps about renting e-scooters. This last category divides the e-scooter batteries in a logical way. This way can be seen in the table 2.

Percentage of the e-scooters	Battery
5	100
25	[75,100)
35	[50,75)
30	[25,50)
5	[15,25)

Table 2: Real e-scooters scenario

Now, we mix different input data so we can see how the algorithm behaves. We will look at the behaviour according to three types; looking at how it reacts depending on the number of goals, the number of nodes that have an e-scooter and finally the battery they have.

It is important to know that when we performed the experiments to look at the behaviour according to the number of goals, the battery... We take the real case for the other categories. An example; we want to look at the battery of e-scooters, we take 3 targets and that 10 percent of the nodes have an e-scooter. The number of goals, either 1, 2 or 3, could logically be a real case, but we choose 3 as this way the algorithm works longer.

- **Goals experiment:** We look at how it behaves according to the number of targets. For each source node, we will make 3 charts, one for each category (number of goals). This way we will check how the algorithm behaves with 1, 2 or 3 goals both in Dijkstra and A*. The e-scooters in this case, will be located at 10 per cent of the nodes and for the battery we will use the one the the table 2, to get closer to a real case.
- **Number of e-scooters experiment:** We will check how the algorithm behaves according to the number of nodes that have e-scooters. We will also have 3 charts, where in each graph we will have the result in Dijkstra and A*. Each chart corresponds to a percentage of nodes with e-scooter, remember that these percentages are; 2, 10 and 20. Then, for each one we will perform the experiment with 3 goals and the battery of e-scooters will be set as shown in table 2.
- **Battery of the e-scooters experiment:** Finally, in this last set we will look at how the algorithm behaves depending on the battery of the e-scooters. Remember that we also had three categories for this; 100 percent battery, 15 percent battery and distributed as shown in table 2. Then, to make it closer to a real case, we will say that we have 3 targets to reach and that only 10 percent of the nodes have an e-scooter. We will obtain three charts in which each one will have the Dijkstra and A* experiments performed.

Once all the experiments have been carried out, what we have to do is to draw the charts. To do this, what we have been doing is for each of the 150 iterations that we have per experiment, we have to save a point.

This point, obviously has part x and part y. The x part would be the total real distance travelled, and the y part is the time taken by the algorithm. We have 150 iterations, each of the 150 iterations has a source node and different targets as we have already explained above. But for each chart, a priori, we have 300 points, 150 in Dijkstra and 150 in A*. We say a priori because those unsolvable experiments, i.e., that cannot reach all the targets, are discarded, we do not keep them as points.

5.1.1 Solution example

But obviously we do not only get the points mentioned above. The main objective of this project is to offer you a solution to a problem. The solution to this problem is to give you the optimal route to reach all your destinations. An example of one of the results obtained is shown in Fig15.

In figure 15, we see the solution to one of the 150 experiments carried out to test the behaviour of the algorithm with 3 goals. In the figure 15 we can see the ID of the experiment, in this case it is experiment number 136. In addition, we can see that in the graph we have 760 e-scooters distributed, remember that for each experiment, the position of the e-scooters is the same. We say, that the battery is "Random", remember what it meant with table 2. It tells you from which node you start and the battery of the e-scooter with which you start the journey. Then, it tells you the places to visit, it tells you the index of the nodes and it gives you a heuristic distance, as a reference, that you will have to do to reach all the goals. Later, it indicates the first node to visit, although it gives you the goal nodes in the order in which you are going to visit them. Now, it indicates all the nodes you have to go through in order to reach all the goals. Note that it tells you when you have to change e-scooter. In the example, we can see that when we reach node 2686, we have changed e-scooter at node 2685. Then, it indicates the real distance travelled and the time taken by the logarithms to find the route. These times are not the ones that are the direct reference to discuss how they behave in the section 5.2, since we do an average of the 150 experiments.

```
ID: 136
Number of e-scooters: 760
Battery of e-scooters: Random
Source: 4546
Battery I start with: 26.382808678253767
Places I want to go in that order:
4748, 4142, 649,
Heuristic Distance: 12246.628351153733 m
First GOAL is: 4748

Pathing of the state that took less time to arrive All goals:
4546, 2683, 2684, 2685, Changed e-scooter -> 2686, 2687, 5181, 5866, 90, 149, 1735, 1754, 6117, 3049, 3055, 1758, 1752, 1751, 1750, 1749,
3051, 4610, 1372, 5255, 1371, 5257, 1367, 1366, 1373, 5379, 7238, 1808, 2667, 1807, 1812, 4442, 4928, 1815, 1817, 4749, 4748, 4749, 2669,
4924, 166, 167, 6202, 2369, 6851, 6244, 7326, 1393, 1398, 7191, 5431, 164, 223, 5783, 5784, 5781, 5782, 7198, 6260, 5556, 5527, 5528, 5087,
4365, 7278, 6235, 4366, 5512, 5957, 762, 4145, 4146, 6473, 3622, 778, 7316, 4580, 4128, 1024, 6610, 812, 781, 4129, 6356, 1464, 6355, 4142,
409, 404, 5894, 5074, 5075, 5076, 407, 996, 397, 1001, 5651, 5851, 379, 5038, 1070, 1071, 6469, 6847, 5844, 370, 369, 4246, 3732, 5841, 5838,
825, 826, 6965, 7400, 5465, 5466, 6465, 5480, 5481, 5483, 5479, 6464, 5031, 7524, 6169, 984, 5028, 5030, 5908, 5909, 629, 3158, 4098, 3159, 4096, 4097, 649,
Time took to arrive all goals: 25.014706666666655 minutes.

Total distance travelled: 15011.02999999999 meters
Time of the algorithm A*: 78
Time of the algorithm Dijkstra: 78
```

Figure 15: Example of a route solution for a random input data

5.2 Results

In this section we will discuss the charts we have obtained from the experiments. We will draw conclusions on how the algorithm works regardless of the input data we introduce. In addition, we will see the difference between Dijkstra and A*.

5.2.1 Commentaries about the charts

In this section we will not yet comment on the time difference between the Dijkstra and A* algorithms. We will make the comparisons separately. We will discuss the results between the different times of the same algorithm depending on the input data. Then, we will compare the times between A* and Dijkstra.

1. **Goals experiment:** We will start by commenting on the experiments carried out in the first set 5.1. The charts obtained from these experiments are: Fig16, Fig17 and Fig18.

   The charts, irrespective of the maximum distance travelled, are quite similar. The times are very small, i.e. the time depends rather little on the distance travelled. Since our chart, although we have chosen this size to be as close as possible to a real case, see the map in figure 13, it is also small enough so that the distances do not have so much weight in the time of the algorithm. Even so, separately, we can see in the 3 graphs how the linear regression increases as the distance gets bigger and bigger.

   Now we are going to look at all 3 charts at the same time. If we try to compare them, it is difficult to find any clear difference. That happens because times are very similar, so at first glance it costs. For that we have table 3. In this table we see the average time of the 150 experiments carried out by table and algorithm, that is, it is the average of all the points that we see in the table, separated by algorithm, we do not mix the times from the 2 algorithms (Dijkstra and A*). In this table we can see the difference between the 3 charts. Although the time difference is low, we also obtain that on average with 3 goals, the algorithm takes 5us longer to offer the solution of the route in case A * than if we only have a single objective. In the Dijkstra case, it is almost 6us apart. We obtain logical results.

Number of Goals:	**1**	**2**	**3**
A*	64.772	67.953	70.146
Dijkstra	66.333	68.8	72.101

Table 3: Time in microseconds of the experiments to look at the behaviour of the A* and Dijkstra algorithms according to the number of goals to be visited.

2. **Number of e-scooters experiment:** From this set of experiments, we have obtained the following graphs Fig20, Fig21 and Fig19.

   Recall that in this set of experiments, the number of targets to reach is the same for all. We chose 3 goals, hence in the 3 charts, the maximum distances reached are similar. With a maximum range of about 30km.

   As in the previous case, we obtain the same type of charts. Here we can see that the linear regressions are ascending. It is true that the time in which the algorithm

takes, is very volatile, being so small, we obtain a very small difference between the times.

If we take a good look at table 4, we can see the times, and we can see that they are reasonable times. They are reasonable times for two main reasons. The fact that the map, even though it is a real scenario, is small means that the number of e-scooters, in a logical scenario, is not very relevant. So, we can get an e-scooter without straying too far from the route. The other point is that the more e-scooters there are, the more states are created, which slows down the search time of the algorithms. With this reasoning, we can understand that the experiment carried out with only 150 e-scooters is the lowest but we can't draw any serious conclusion because the differences are too small.

Number of E-scooters:	**150**	**760**	**1520**
A*	69.373	70.445	69.763
Dijkstra	70.926	71.899	72.273

Table 4: Time in microseconds of the experiments to look at the behaviour of the A* and Dijkstra algorithms according to the number of e-scooters.

3. **Battery of the e-scooters experiment:** From this last set of experiments we obtain the following charts: Fig24, Fig23 and Fig22.

   Looking at the graphs, we can draw very similar conclusions to those of experiment set number 2, as they also have 3 goals, and the distance range is very similar. That makes, that the algorithm times are quite similar.

   Let's look at table 5. In this table we see how the times of the algorithms change with respect to the battery that the e-scooters have. We can clearly see that the fact that all the e-scooters have 100 percentage of the battery means that there are fewer changes of e-scooters, i.e. we save time when searching for e-scooters to reach all the targets, so the search time of the algorithms is shorter. Looking at the times of the "Random" batteries, and when everyone has 15 percentage of battery they seem to me more rare. It doesn't fit very well since "Random" would have to be lower, but it is true that in the 150 experiments carried out in "Random" it could be that the route passed through places with e-scooters with a low battery. Even so, the time between these two categories is very small, the search time is still fast.

Battery of the e-scooters:	**Random**	**100**	**15**
A*	70.12	68.886	69.306
Dijkstra	71.233	70.32	71.06

Table 5: Time in microseconds of the experiments to look at the behaviour of the A* and Dijkstra algorithms according to the battery of the e-scooters.

To conclude this section, I would like to make an aside on the comparison of the algorithms. We have observed that in all the experiments carried out, the time of the A* algorithm is less than that of Dijkstra. This is because, as we have already learned, Dijkstra has to go through almost all the nodes to find the solution, it has no heuristic distance. But it is true that for two reasons, the time difference is not so great. This is because the map is not big enough to have a significant weight and also because for the creation of each state, A* has to find a heuristic distance. This search requires a computation that slows down the algorithm, but Dijkstra saves it because the heuristic distance for it is 0, it does no computation.

5.2.2 Charts

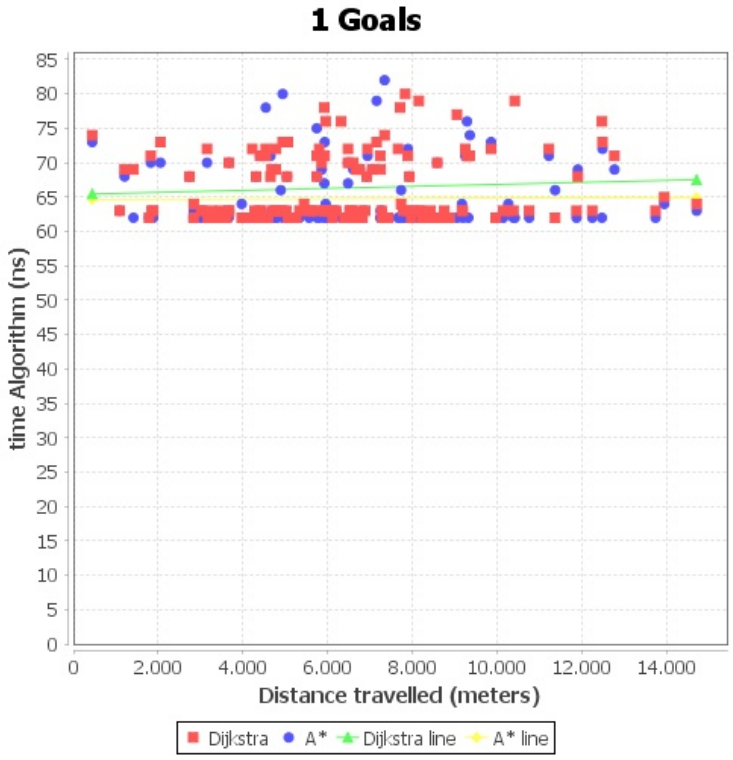

Figure 16: 1 Goal

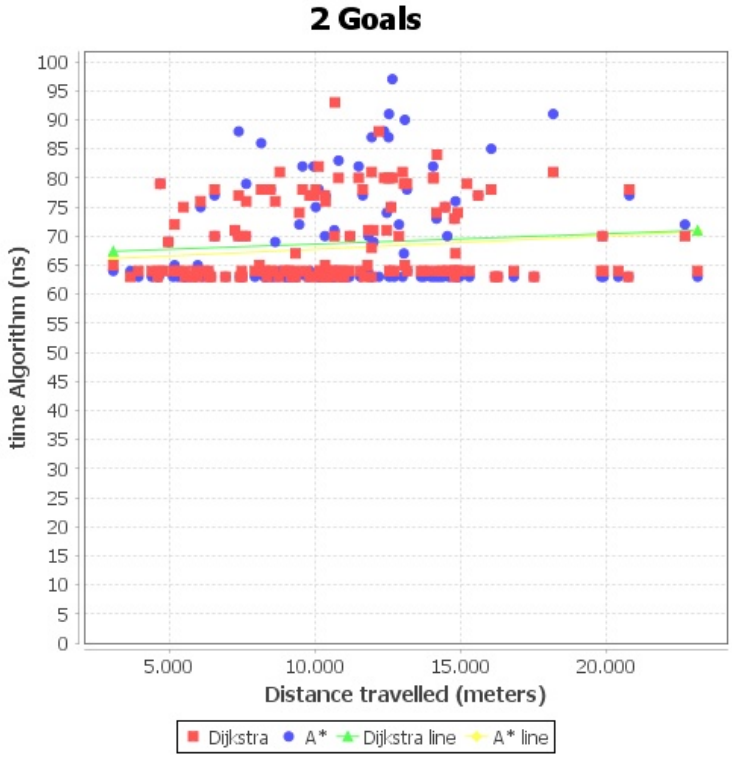

Figure 17: 2 Goals

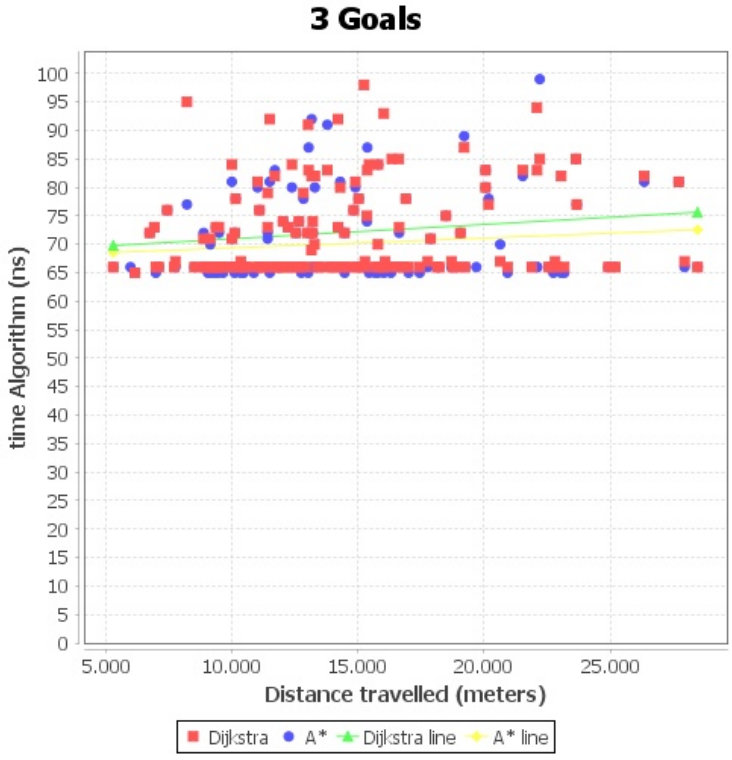

Figure 18: 3 Goals

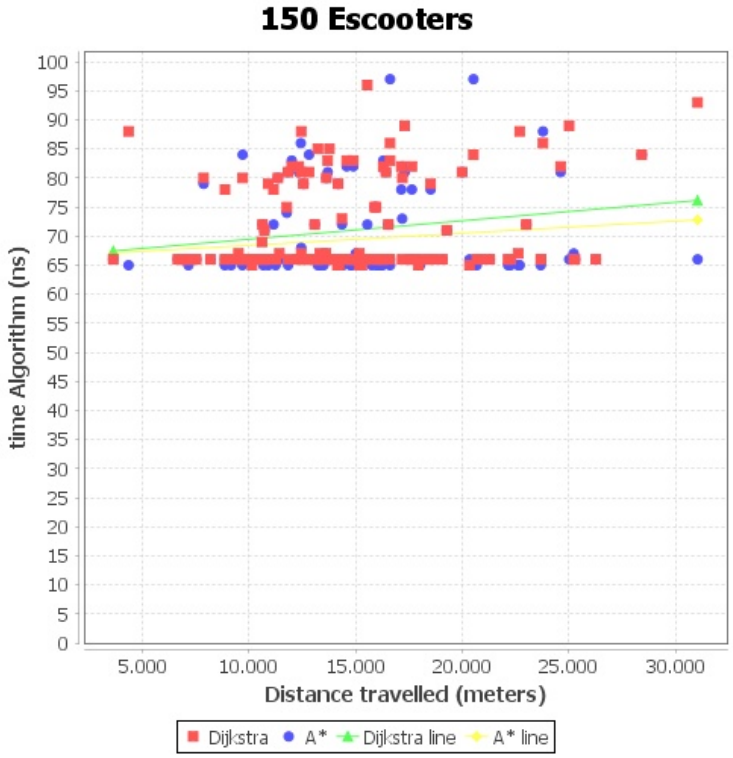

Figure 19: 150 e-scooters

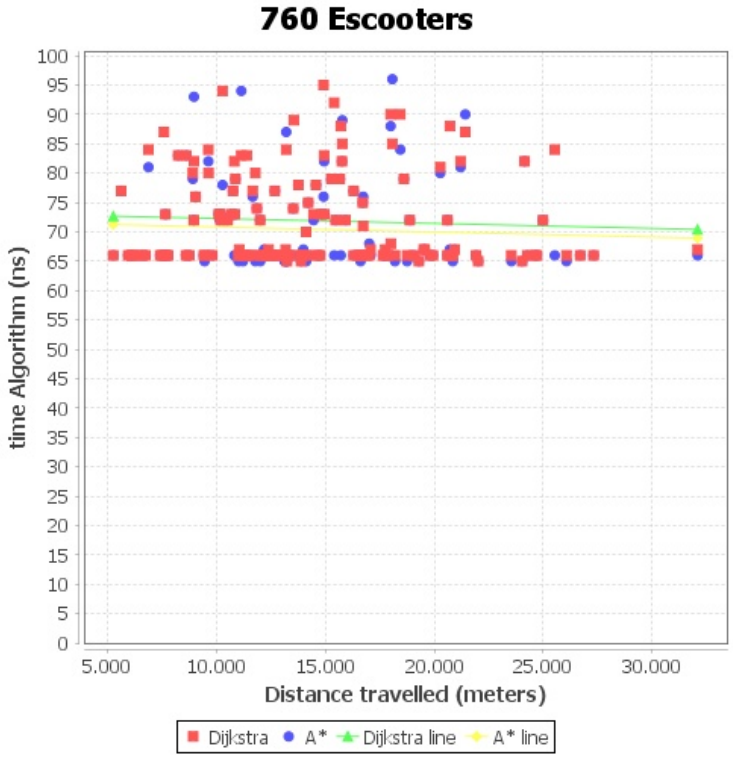

Figure 20: 760 e-scooters

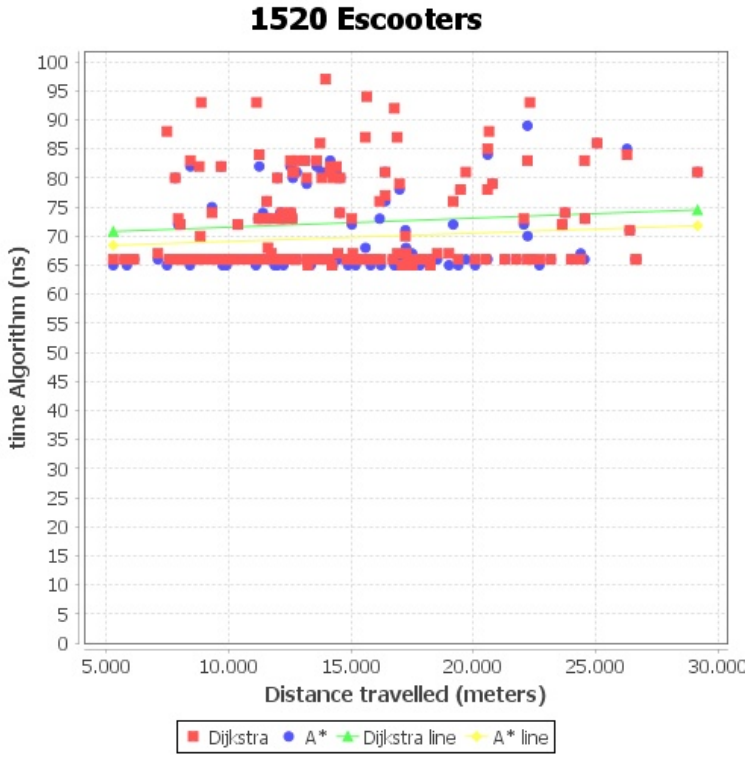

Figure 21: 1520 e-scooters

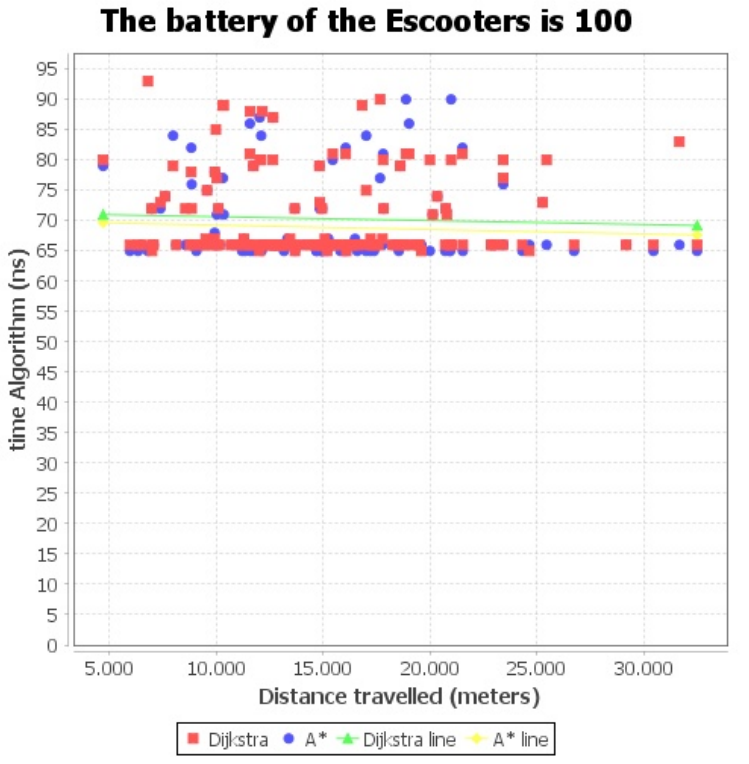

Figure 22: 100 battery

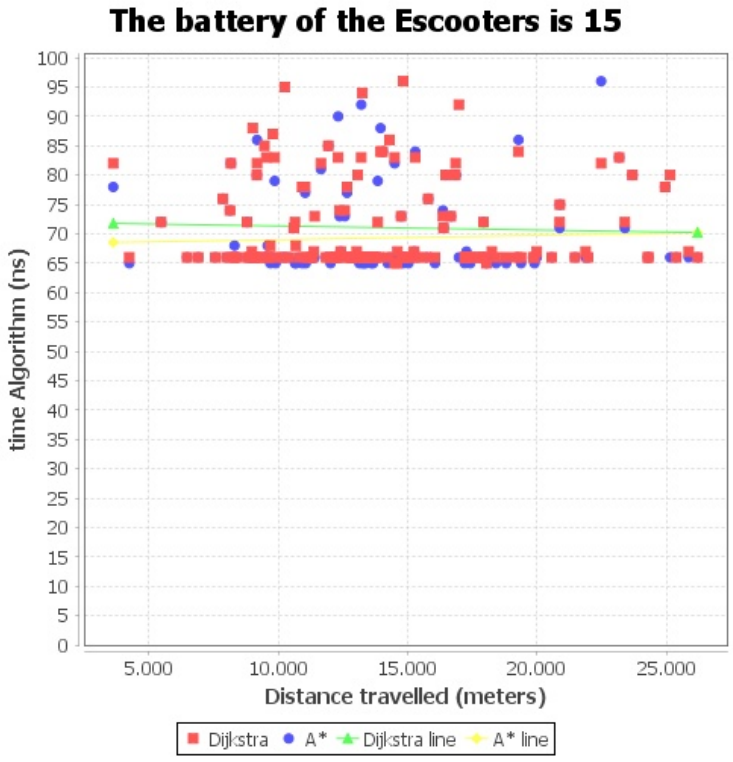

Figure 23: 15 battery

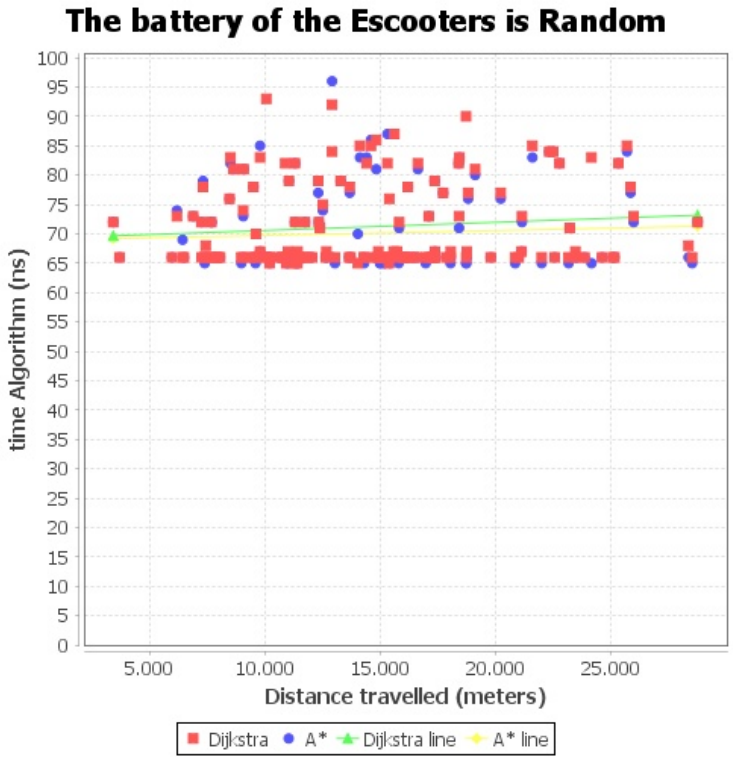

Figure 24: Random battery

6 Budget

The budget of this project is:

- Computer: An ASUS computer, the original cost of which was 799€
- Salary: A student of Telecommunications Engineering at the ETSETB is paid 9€/h when he/she is in a company doing a curricular internship. The total hours are 450.
- At Software level. All the software used for this project, such as QGIS, NetBeans IDE and Sublime Text 3 are Open Source.

Items	Concept	Amount(€)
Item 1	Computer	799
Item 2	Salary	4050
Item 3	Software	0
Total		4849

Table 6: Budget

The total cost is 4849€.

7 Conclusions

In this section we will draw conclusions from the completed project. We will discuss what we have learned and whether the results obtained are satisfactory.

It has been a project, which in my opinion has been very comprehensive. It has taken many hours of work but it has been worth it in the end. We have been able to achieve the objectives we had set ourselves. We managed to create a route planning problem that could perfectly well be a real case. But in addition, we were able to solve this problem. We have managed to create a fairly fast algorithm that finds the fastest route (optimal route) in a short time.

We have learned different data structures in Java, and after some study, we have been able to create the best one for our case. In addition, we have also learned about search algorithms.

On the A* vs Dijsktra comparison. We have seen in real cases how A* is faster than Dijsktra. So it was a good decision to base my algorithm on A*. In addition, the fact of doing it with states, has made me optimise the search time, based on the heuristic distances.

As expected, we could see how the algorithm time increases with the number of targets. But what we have noticed is that in maps, like ours, in a real case like the city of Prague, the number of e-scooters and the battery in logical situations is not very relevant. The search time does not change that much.

This leads us to think about the following. Let's imagine that an e-scooter company wants to add software with similar functionality to this project to their app. This company would only have to take into account the number of targets to be reached. The aspects of the number of e-scooters, being a reasonable number, a real case and their respective batteries do not have much influence on the route finding. Then they would have to focus more on finding an optimised way for the software to give them the solution depending on the targets to be reached.

8 Future Work

In this section we will discuss improvements that can be made from this project. These improvements could not be made in this project for various reasons such as lack of time, lack of knowledge or other problems.

All of the following ideas are ideas that could help this project come closer to a real case.

For example, right now the battery reduction is linear, it only depends on the distance travelled. It would be possible to make the battery reduction also depend on other factors such as the weight of the user, if the terrain is sloping, the road surface. In addition, the actual speed at which the user is going at any given moment could also be taken into account as a factor in the reduction of the battery. These are factors that would help the battery reduction to be closer to a real case.

It would be nice to have a collaboration with an e-scooter rental company. So that they could provide us with real data on the actual positioning of the e-scooters and their respective batteries.

Moreover, right now, if you can't reach all your goals on an e-scooter, that scenario is marked as unsolvable. One last idea would be that when this happens, the user would start walking to the nearest e-scooter or directly to the next target. As long as the optimal route is chosen, but in this way all scenarios can be solved.

Finally, a very good idea to show the solution. This solution would be to create a graphical interface that would show you the route and changes of the e-scooter through an interactive map.

References

[1] Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser. *Data structures and algorithms in Java*. John Wiley & Sons, 2014.

[2] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[3] Michael Main and M Main. *Data structures & other objects using Java*. Addison-Wesley, 2003.

[4] Yong Deng, Yuxin Chen, Yajuan Zhang, and Sankaran Mahadevan. Fuzzy dijkstra algorithm for shortest path problem under uncertain environment. *Applied Soft Computing*, 12(3):1231–1237, 2012.

[5] Huijuan Wang, Yuan Yu, and Quanbo Yuan. Application of dijkstra algorithm in robot path-planning. In *2011 second international conference on mechanic automation and control engineering*, pages 1067–1069. IEEE, 2011.

[6] Masoud Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. Investigation of the*(star) search algorithms: Characteristics, methods and approaches. *World Applied Programming*, 2(4):251–256, 2012.

[7] Lawrence Mandow, JL Pérez De la Cruz, et al. A new approach to multiobjective a* search. In *IJCAI*, volume 8. Citeseer, 2005.

[8] Amir R Soltani, Hissam Tawfik, John Yannis Goulermas, and Terrence Fernando. Path planning in construction sites: performance evaluation of the dijkstra, a, and ga search algorithms. *Advanced engineering informatics*, 16(4):291–303, 2002.

[9] David L Applegate, Robert E Bixby, Vašek Chvátal, and William J Cook. *The traveling salesman problem*. Princeton university press, 2011.

[10] Liping Fu, Dihua Sun, and Laurence R Rilett. Heuristic shortest path algorithms for transportation applications: state of the art. *Computers & Operations Research*, 33(11):3324–3343, 2006.

Abbreviations

CTU Czech Technical University

ETSETB Barcelona School of Telecommunications Engeneering

FIFO First In First Out

LIFO Last In First Out

OSM Open Street Maps

TSP Travel Salesman Problem

VRP Vehicle Route Problem