U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Faculty Of Engineering, University Of Porto

Informatics and Computing Engineering Department

# MyJourney!

Algorithm Design and Analysis

*Authors*
André Moreira, 201904721
Nuno Alves, 201908250
Nuno Costa, 201906272

May 24, 2021

# Contents

# Introduction

Parking management in a big city can be quite challenging. Efficiently managing availability, time and cost is a tough task, even when all the information is gathered.

Furthermore, people don't always know where the parking spots are located and which ones are available, which may result in congested city traffic, parking allocation inefficiency or time wasted in the inevitable task of finding a parking place.

The problem that will be analysed in this project derives from the Travelling Salesman Problem. Given the origin and destination locations, a suitable parking spot, as well as the path to it that goes through all chosen points of interest, should be found. For this, the chosen solution must account for a set of variables and restrictions. The chosen parking spot must be available, close to the destination location - within reasonable walking distance -, should be as cheap as possible and the path's distance must also be as short as possible.

However, meeting all these conditions and obtaining the best solution possible can become time costly, especially as the problem's size grows. Time complexity is an important measure to calculate the viability of an algorithm and should be also accounted for.

With all of this in mind, the following report seeks to solve this problem and all of its intricacies, aiming for a real-life solution to the problem.

# Problem Definition

In this chapter the problem is defined in terms of input data, output data, its constraints and the objective function.

## 2.1 Input Data

All the needed information to compute a solution for the problem is defined below. It should be noted that the following structures represent the input data needed - additional parameters may be added to the structures defined below for the algorithms that need them.

### 2.1.1 Definition

1. The map should be represented by a directed graph $G = (N, E)$, where
   (a) $N$ is the set of nodes of the graph.
   (b) $E$ is the set of edges of the graph (where the edges represent any given connection between two nodes, being it a street, a highway, etc.).
2. $N_k \in N$, $1 \leqslant k \leqslant |N|$ is a node of the graph represented by a tuple $N_k = (la, lo, A, P_{info})$ where
   (a) $la(N_k)$ is the latitude of the node.
   (b) $lo(N_k)$ is the longitude of the node.
   (c) $A(N_k)$ is the set containing edges connected to the node, such that $A \subseteq E$.
   (d) $P_{info}(N_k)$ is the parking information associated to that node.
      i. If N is a parking lot, $P_{info}(N_k)$ represents a parking lot $P_k = (c, c_{max}, f) \in P$, $1 \leqslant k \leqslant |P|$ where
         A. $c(P_k)$ is its current capacity.
         B. $c_{max}(P_k)$ is its maximum capacity.
         C. $f(P_k) = f(t, o)$ is the parking fee function, which can vary both according to the parking time $t_k \in t_{parking}$ and the occupancy rate $o = \frac{c}{c_{max}}$.
      ii. If N is not a parking lot, $P_{info}(N_k) = (r_{parking})$, where

A. $r_{parking}$ is a boolean that defines whether or not this node requires nearby parking, in case of it being a POI for the user.

3. $E_k \in E$, $1 \leqslant k \leqslant |E|$ is an edge of the graph represented by a tuple $E_k = (N_{origin}, N_{destination}, w)$ where
   (a) $N_{origin}(E_k) \in N$ is the origin node of the edge.
   (b) $N_{destination}(E_k) \in N$ is the destination node of the edge.
   (c) $w(E_k)$ is the weight of the edge which constitutes its travel distance.
4. $O$ is the origin node, $O \in N$.
5. $D$ is the destination node, $D \in N$.
6. $POI$ are the nodes/points of interest which the traveler must visit before reaching node $D$, $POI \subseteq N$.
7. $t_{parking}$ is the array with the parking times of each POI/Destination.
8. $d_{threshold}$ is the distance the user is willing to walk at most.

### 2.1.2 Constraints

1. $\forall\, N_k \in N :$
   (a) $-90° \leqslant la(N_k) \leqslant 90°$
   (b) $-180° \leqslant lo(N_k) \leqslant 180°$
2. $\forall\, N_k \in N, \exists\, E_k \in E : N_{origin}(E_k) = N_k \vee N_{destination}(E_k) = N_k$
3. $\forall\, E_k \in E : w(E_k) > 0$
4. $\forall\, P_k \in P : \{P_k = P_{info}(N_k)$ where $N_k$ is a parking lot$\}$,
   (a) $c_{max}(P_k) > 0$
   (b) $0 \leqslant c(P_k) \leqslant c_{max}(P_k)$
   (c) $\forall\, t \geqslant 0 : f(P_k)(t, o) \geqslant 0$
5. $O \neq D$
6. $\forall\, t_k \in t_{parking} :\ t_k > 0$
7. $|t_{parking}| = |POI| + 1$, which means that every park chosen must have a parking time associated to it (for each POI and destination).
8. $d_{threshold} > 0$.

## 2.2 Output Data

The output of a solution is described below.

### 2.2.1 Definitions

1. The ordered list of paths $S$ that, for each $S_k$, $1 \leqslant k \leqslant |POI| + 1$, $S_k = (S^v, S^f)$, where
   (a) $S^v$ is the ordered list of travelled edges that form the solution path from the origin/last POI node to the parking node $k$ of the POI/destination.
   (b) $S^f$ is the ordered list of travelled edges that form the solution path from the parking node to the respective POI/destination node.
2. A simplified graph $G_s = (N_s, E_s)$ containing only the nodes and edges that are present in the solution paths $S$, where $N_s \in N \wedge E_s \in E$.
3. The chosen parking lots array $P_{chosen} \subseteq N$.
4. Distance taken by vehicle $d_v = \sum_{i=1}^{|S|} \sum_{k=1}^{|S^v(S_i)|} w_k(S^v(S_i))$.
5. Distance taken by foot $d_f = \sum_{i=1}^{|S|} \sum_{k=1}^{|S^f(S_i)|} w_k(S^f(S_i))$.
6. The total price of the parking $p$ given by the function $f(P_k) = f(t, o)$ for each parking such that $p = \sum_{k=1}^{|P_{chosen}|} f(t_k, \dfrac{c(P_k)}{c_{max}(P_k)})$.

### 2.2.2 Constraints

1. Travelled edges $S^w$
   (a) The first edge $E_0$ of the travelled edges list, $N_{origin}(E_0)$ must be node $O$.
   (b) The last edge $E_{last}$ of the travelled edges list, $N_{origin}(E_{last})$ must be node $P_{chosen}$.
2. Travelled edges $S^f$
   (a) The first edge $E_0$ of the travelled edges list, $N_{origin}(E_0)$ must be node $P_{chosen}$.
   (b) The last edge $E_{last}$ of the travelled edges list, $N_{origin}(E_{last})$ must be node $D$.
3. Distance traveled by foot $d_f \geqslant 0$
4. Distance traveled by vehicle $d_v \geqslant 0$

## 2.3 Objective Function

The problem is, in fact, complex. Taking a bottom-up approach on it, one can break it up into 3 core ideas (and 3 distinct objective functions).

Firstly, an available parking lot near the destination node which minimizes both distances and cost should be found. For that, we can define a function such that:

$$cost_1(d_f, p) = k_f \times d_f + k_p \times p \tag{2.1}$$

where:

$d_f$ = the distance travelled by foot
$p$ = the parking price
$k_f$ = the foot travel distance importance coefficient
$k_p$ = the parking price importance coefficient

The intention is to minimize the cost. As such, the objective function should be $min(cost_1(d_f, p))$. However, it must be noted that this minimum is dynamic - it may very well vary according to the user preferences, or in other words, according to the foot travel distance and price importance coefficients. This function will be 0 if $r_{parking}$ is true for the destination node.

Secondly, the path between the chosen parking lot and the origin should be defined. A cost function for that problem can be defined as:

$$cost_2(d_v) = d_v \tag{2.2}$$

where:

$d_v$ = the distance travelled by vehicle to the parking lot (or destination if $r_{parking}$ is true)

As explained above, the intention is to minimize the cost, which means that the objective function is also defined by $min(cost_2(d_v))$. This can be simplified to $min(d_v)$.

Lastly, we need to join both this approaches and apply them to each origin-POI, POI-POI or POI-destination (origin-destination if no POI exist) path. There's two ways to look at the problem given: either the user already gives an order to search each POI towards the destination node, or the application must be capable of choosing a fitting order (as it must pass through all of them).

If the first situation is at hand, the problem is reduced to applying the first two objective functions to each pair.

However, that may not be the case. As such, a third objective function must be defined. Since calculating the overall minimum distance for a TSP-like problem would be extremely costly, a shift in perspective must be made. As such, one might simply consider the POIs given, the origin and the destination nodes as one smaller graph, where its edges are defined by the distance between them on the map. With that smaller graph, calculating a TSP-like problem seems feasible. As such, defining $d_{tsp}$ as the distance in that TSP-like sub problem, the third and final objective function can be defined as:

$$cost_3(d_{tsp}) = d_{tsp} \tag{2.3}$$

where:

$d_{tsp}$ = the total path length of the tour of all nodes of the smaller graph

As before, the intention is still to minimize the cost, which leads to the third and final objective function being $min(cost_3(d_{tsp})) = min(d_{tsp})$.

To conclude, the solution must, as such, minimize the first two functions for each pair on the minimized tour defined by the third objective function [1].

[1]It should be clear, however, that these functions should be minimized in reasonable running time, which may not lead to the absolute minimum value, but a reasonable near-minimum value instead.

# Prospective Solution

## 3.1 Algorithm Overview and Analysis

Due to the nature of this problem, it can be divided into three parts (even though they are not independent), as explained before.

Firstly, one can try to find available parking spots within a reasonable distance of the destination node using a Single Source Shortest Path (SSSP) algorithm. From those found parking spots, a cost evaluation occurs, selecting only one.

Secondly, having the selected parking spot defined, a simple Shortest Path algorithm can be used to find the shortest path between that and the origin node. With that path defined, the solution for the sub-path of the total tour is found. As such, this solution must be applied to all sub-paths of the tour, which is defined either by input or by using a TSP solving algorithm. When all sub-paths are solved, the complete solution is finally defined.

However, it is important to firstly check that the graph given is connected, so that it is assured that a solution can be found.

### 3.1.1 Verifying Graph's Connectivity

For this solution, it is important that the given graph is strongly connected. In fact, a weakly connected graph would not be enough for the purposes of this problem since it would be possible to select a node that had no path to another selected node. Even if the given graph was unilaterally connected, it wouldn't be enough since the user should have the liberty to travel the opposite direction. As such, a strongly connected graph, which has a path between any pair of nodes (u,v), both from u to v and from v to u, must be given as input.

To verify this condition, a pre-processing algorithm must be run on the graph.

#### Strongly Connected Components

A SSC (short for Strongly Connected Component) is defined as a graph's cycle in which any node can reach any other node in the same SSC.

As explained previously, we must guarantee that the graph is strongly connected. A smart way to do that is to assure that the graph has one and only one SSC. To do this, we can use a SSC algorithm.

There are several algorithms for this purpose, such as the Kosaraju's algorithm and Path-Based strong component algorithm, but we'll focus on the Tarjan's algorithm, due to its $O(|V| + |E|)$ linear time complexity.

### Tarjan's Algorithm

To understand Tarjan's algorithm, one must first understand the concept of **low-link values**. A low-link value of a node is the smallest node ID reachable when doing a Depth First Search algorithm from that same node (including itself).

After calculating each node's low-link, it is possible to infer what SSC exist in a graph, since every node in a given SSC will have the same low-link value.

However, this might not be true for every case. This low-link calculation for each node is highly dependent on the Depth First Search algorithm. Diferent traversals will most likely result in diferent results, which can't be true since an unchanged graph should still have the same SSC, no matter the traversal [1].

To fix this flaw, the Tarjan's algorithm uses what is called the **stack invariant**. This invariant keeps the valid nodes from which to update low-link values from. As such, nodes are added when they are first encountered and removed when a new SCC is found. With this, the SCC aren't 'contaminated' by other SCC upon the Depth First Search since it is possible to check whether or not the node we're currently comparing to belongs to an already closed SCC (that is, if it isn't on the stack).

The pseudo-code for this algorithm can be defined as follows:

**Algorithm 1** Tarjan's Algorithm Adaptation

```
1: $sccs \leftarrow 0$
2: $currentID \leftarrow 0$
3: $stack \leftarrow emptystack()$
4:
5: for $node \in N$ do
6:     $visited(node) \leftarrow false$
7: end for
8: for $node \in N$ do
9:     if not $visited(node)$ then
10:        $dfs(node)$
11:    end if
12: end for
13: return $sccs$ equals 1
```

**Algorithm 2** DFS function

```
1: $push(stack, node)$
2: $id(node) \leftarrow currentID$
3: $lowlink(node) \leftarrow currentID$
4: $currentId \leftarrow currentID + 1$
5: $visited(node) \leftarrow true$
6:
7: for each $adj \in adj(node)$ do
8:     if not $visited(adj)$ then
9:         $dfs(adj)$
10:    end if
11:    if $adj \in stack$ then
12:        $lowlink(node) \leftarrow min(lowlink(adj), lowlink(node))$
13:    end if
14: end for
15:
16: if $id(node)$ equals $lowlink(node)$ then
17:     while $pop(stack)$ differs from $node$ do
18:     end while
19:     $sccs \leftarrow sccs + 1$
20: end if
```

With the return value of the previously shown function, it is possible to confirm whether or not the graph is strongly connected. It must be noted, however, that this algorithm opens up possibilities for the solution: if the graph given by input is not strongly connected, but if the nodes given as Origin, points of interest and Destination by the user belong to the same SCC, the algorithm can still run normally (assuming that the SCC has parking spots, which is problem that may be detected later on). This will be taken into consideration upon the implementation.

### 3.1.2 Selecting the best parking spot

As mentioned, an SSSP algorithm must be used for this part of the solution. For that, a Dijkstra Shortest Path algorithm was chosen (with a time complexity of $O(|E| + |V|log(|V|))$ - using a priority queue).

In order to fulfill the needs of the problem at hand, the algorithm must be modified. As previously mentioned, the user must set a threshold that defines the maximum distance he's willing to walk. The use of a priority queue makes it easier to know when that threshold was surpassed (by definition, we will work with a strictly increasing distance between the source node and current node).

However, setting only one threshold might not be correct. In a less crowded place, one might find a lot of parking spots in just under 300 meters of the destination, while in some other places with great turnout, there might be fewer available spots throughout a much greater area. As such, to better solve these cases, two thresholds were set, according to the user's max threshold choice.

The following algorithm pseudo-code solves the presented problem.

**Algorithm 3** Dijkstra Adaptation

```
1: for each $vertex \in V$ do
2:     $dist(vertex) \leftarrow \infty$
3:     $prev(vertex) \leftarrow nil$
4:     $visited(vertex) \leftarrow false$
5: end for
6: $dist(origin) \leftarrow 0$
7: $pq \leftarrow makequeue()$
8: $push(pq, origin)$
9:
10: $parks \leftarrow \{\}$
11: $currdist \leftarrow 0$
12: $threshold1() \leftarrow return\ currdist < max1$
13: $threshold2() \leftarrow return\ size(parks) < minparks\ and\ currdist < max2$
14:
15: while $pq$ is not empty $and$ $(threshold1()\ or\ threshold2())$ do
16:     $current \leftarrow popmin(pq)$
17:     $currdist \leftarrow dist(current)$
18:     $visited(current) \leftarrow true$
19:     if $current$ is parking and not $c(current) = c_{max}(current)$ then
20:         $insert(parks, current)$
21:     end if
22:     for each $edge \in adj(current)$ do
23:         $to \leftarrow to(edge)$
24:         if $dist(to) > dist(current) + weight(edge)$ then
25:             $dist(to) \leftarrow dist(current) + weight(edge)$
26:             $prev(to) \leftarrow current$
27:             if $not\ visited(current)$ then
28:                 $push(pq, to)$
29:             else
30:                 $decreasekey(pq, to)$
31:             end if
32:         end if
33:     end for
34: end while
35: return $parks$
```

With the surrounding parks selected, it is key to filter them down. This filter takes into account the distance from the destination node and the price it will charge the user. The distance from the origin shouldn't be a part of this filter: in fact, if a parking spot farther from the origin node is chosen, it means its walking distance to the destination node and price were lower than its counterparts, and the relatively small distance difference (because of the set threshold) shouldn't be accounted for.

In fact, more than one could be chosen, but this adaptation only makes sense if, in fact, an unpredicted inefficiency on the cost evaluation occurs. In theory, the best parking spot on this phase should still be the best in the following, since the path to it does not impact this decision.

After the applied filter and the chosen parks selected, the algorithm can now proceed to the second part.

### 3.1.3 Shortest path to parking spot

With a best park selected, we can now apply a Shortest Path algorithm to select the best (or an approximation of the best) path from the origin node to that parking spot. For that, the A* algorithm was chosen (an adaptation of Dijkstra's with a heuristic to improve time complexity).

#### A* Heuristic Definition

Since the origin and destination nodes are known, we can define a total distance approximation function $f(node)$, which is defined as

$$f(node) = g(node) + h(node) \tag{3.1}$$

where:

$g(node) =$ the distance from the origin node to the current node
$h(node) =$ an approximation of the distance from the current node to the destination node

This approximation distance can be calculated in one of many ways. An easy way to do that would be to consider the remaining distance to the destination node to be the straight line that connects the current node and the destination node. In fact, choosing this distance will make A* always achieve the best solution possible, since it is never an overestimating heuristic of the real remaining distance (as proven in [2][3], it is an *admissable* heuristic). However, to make a straight line length calculation, one must use the square root operation, which may not be as efficient. As such, one can use the Manhattan distance, which can be an overestimation of the result in this problem (since it is not a grid map, the user can in fact travel in the shortest diagonal between the two points), and may in fact not lead to the most optimal solution.

With this in mind, it possible to define several distance functions. Since we're given (latitude, longitude) pairs for each node, and the calculation of geodesic distances with the Haversine or Great-circle distance functions is not properly efficient, a variation was found. As such, the straight line distance function can be defined as ($d_{meridian}$ and $d_{equator}$ explained in [4]):

**Algorithm 4** Distance Function - Haversine variation

```
1: $dx = d_{meridian}\frac{|lat_1 - lat_2|}{180}$
2: $dy = d_{equator}\frac{|lng_1 - lng_2|}{180}cos(\frac{lat1+lat2}{2})$
3: **return** $\sqrt{dx^2 + dy^2}$
```

As mentioned previously, this could be adapted to use a Manhattan distance, where the use of the square root is ignored, and the sum of the x and y variations is returned instead.

**Algorithm 5** Distance Function - Haversine variation with Manhattan adaptation

```
1: $dx = d_{meridian}\frac{|lat_1 - lat_2|}{180}$
2: $dy = d_{equator}\frac{|lng_1 - lng_2|}{180}cos(\frac{lat1+lat2}{2})$
3: **return** $dx + dy$
```

It must also be noted that if given grid coordinates, both of this functions can be adapted to be more efficient. With that, the cosine could be ignored and a simple Euclidean distance or, for even more efficiency, a simple Manhattan distance could be calculated.

### A* implementation

With the heuristic defined, it is now possible to define the algorithm implementation. It must be noted that the implementation which will be described here is a simple A* approach for the problem at hand. Other performance enhancers, such as the bidirectional A* approach, Weighted A* approach or the use of Fibonacci Heaps can and will be taken into consideration in the implementation of the solution.

**Algorithm 6** A* algorithm

```
1: **for each** $vertex \in V$ **do**
2:     $dist(vertex) \leftarrow \infty$
3:     $prev(vertex) \leftarrow nil$
4: **end for**
5: $dist(origin) \leftarrow 0$
6: $pq \leftarrow makequeue()$
7: $f_{node} \leftarrow dist(origin) + h_{distance}(origin, destination)$
8: $push(pq, origin, f_{node})$
```

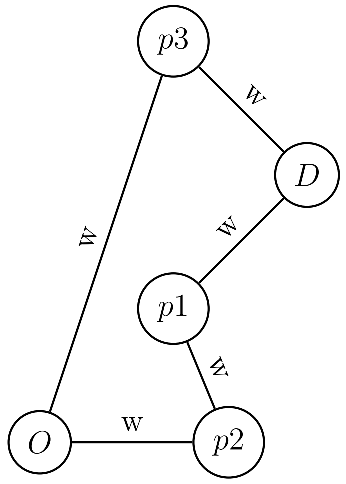
```
 9: while $pq$ is not empty do
10:     $current \leftarrow popmin(pq)$
11:     $closedset \leftarrow closedset \cup \{current\}$
12:     if $current$ is destination then
13:         return $buildpath(origin, current)$
14:     end if
15:     for each $edge \in adj(current)$ do
16:         $to \leftarrow to(edge)$
17:         $cost \leftarrow dist(current) + weight(edge)$
18:         if $dist(to) = \infty$ or $dist(to) > cost$ then
19:             $dist(to) \leftarrow cost$
20:             $prev(to) \leftarrow current$
21:             $f_{node} \leftarrow dist(to) + h_{distance}(to, destination)$
22:             if $to \in pq$ then
23:                 $decreasekey(pq, to, f_{node})$
24:             else
25:                 $push(pq, to, f_{node})$
26:             end if
27:         end if
28:     end for
29: end while
30: return $FAILURE$
```

Note the importance of lines 24-28: since the heuristic used can be *non admissable*, a node may need to be revisited and its distance recalculated, unlike in Dijkstra's algorithm implementation. It should also be noted that the priority queue is defined by the $f(node)$ value previously mentioned.

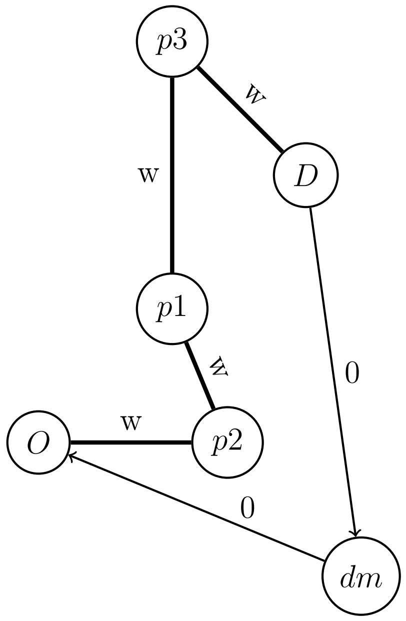
## 3.2 Travelling Salesman Problem Adaptation

The *Travelling Salesman Problem* is one of the most famous NP-Hard problems[5]. Obtaining the optimal solution to this problem is possible with brute force by finding all of the possible routes and selecting the shortest one. Despite generating the optimal solution, its time complexity is $O(n!)$ [6], being n the number of nodes in the set, an unpractical solution. As such, a feasible solution to this problem can be achieved either using heuristics (where the solution found might not be optimal) or reducing time complexity with dynamic programming, for example.

This problem can be adapted to the TSP since the user needs to find the best way to travel through a series of points of interest, with the restriction that the user chooses the first and last node. A possible solution to this restriction is to insert a node ($dm$) in the graph, and two uni-directed edges with *weight* 0: one from the destination to the *dummy* node and the other from the *dummy* to the first node.



(a) TSP without dummy node



(b) TSP with dummy node

Figure 3.1: Adaptation to the TSP

### 3.2.1 Held-Karp Algorithm

The *Held-Karp Algorithm* is an algorithm that solves the TSP using dynamic programming. This algorithm has a better time complexity comparing to the brute force solution, since it presents a $O(n^2 2^n)$ time complexity, which becomes better after some nodes. [7] However, we can verify that it still is exponential which doesn't make it a great solution, particularly if we have a large quantity of nodes.

| Points of Interest | Brute Force | Dynamic Programming |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 2 | 16 |
| ... | ... | ... |
| 6 | 720 | 2304 |
| 7 | 5040 | 6272 |
| ... | ... | ... |
| 17 | 355687428096000 | 37879808 |
| 18 | 6402373705728000 | 84934656 |
| 19 | 121645100408832000 | 189267968 |

Table 3.1: Brute force VS dynamic programming time complexity

### 3.2.2 Heuristics

There are some heuristics that minimize the time complexity of the problem with very good approximations of the optimal solution.

#### Nearest neighbour

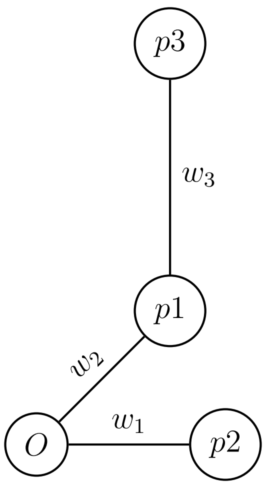
This heuristic applies some sort of a greedy methodology and tries to find the best route by always choosing the next point in the tour to be the closest one, by starting in the $O$ node. Since we just want to approximate the best tour so that after we can apply other path finding algorithms, we'll just use the geographic distance between the nodes.

**Algorithm 7** Nearest neighbour

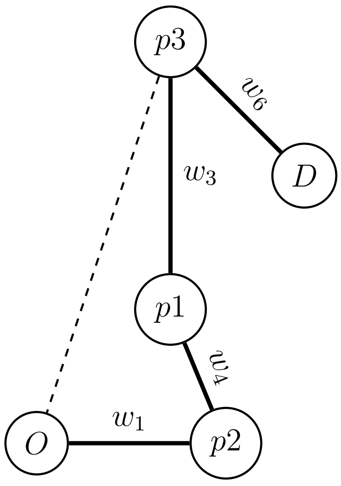
```
1: $tour \leftarrow \{O\}$
2: $last \leftarrow O$
3: while $poi$ not empty do
4:     $minDist \leftarrow \inf$
5:     $next \leftarrow nil$
6:     for each $node \in poi$ do
7:         if $dist(last, node) < minDist$ then
8:             $minDist \leftarrow dist(last, node)$
9:             $next \leftarrow node$
10:        end if
11:    end for
12:    $push(tour, next)$
13:    $last \leftarrow next$
14:    $poi \leftarrow poi \setminus \{next\}$
15: end while
16: $push(tour, D)$
```

### Minimum Spanning Trees

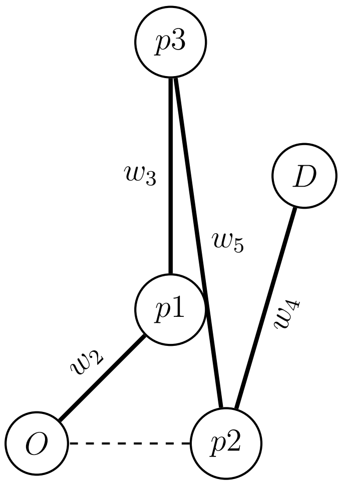
Another possible heuristic is based on using minimum spanning trees. This technique usually finds a result - in this case the distance - 15% to 20% above the optimal solution, which in many applications is enough. This method is based on finding a minimum spanning tree and visiting the resulting tree with a depth first search. [5]. However, in our case, the graph has directed edges, because of the dummy node, so we can't use the strategy described in Figure 3.1 with this heuristic. We can however approximate and find a minimum spanning tree, leaving out the destination node, adding it at the end of the result, as done on the approach for the nearest neighbour solution.



(a) Possible Minimum Spanning Tree



(b) MST after DFS visit, adding D after the last node of the cycle



(c) Another possible approximation, visiting in a different order

Figure 3.2: TSP with minimum spanning tree heuristic

This heuristic can be applied with the following algorithm:

**Algorithm 8** TSP approximated using MST's

```
function $dfsVisit(tour, current)$
    $push(tour, current)$
    for each next in current do
        $visit(tour, next)$
    end for
end function
$tour \leftarrow \{\}$
$tree \leftarrow generateMst(O)$ // Either Kruskal's or Prim's algorithm, with O as root
$dfsVisit(tour, O)$
$push(tour, D)$
```

### 3.2.3 Defined Solution

As exposed, this problem has several solutions. However, all of them accuse a problem: where one lacks time and memory complexity, the other lacks solution accuracy relative to the best solution. As such, a way to minimize this problem is to use the solutions where they really do thrive.

For that, one can use a dynamic programming approach for a small number of POI, as there is no need to compromise the result when the time and memory complexities are still small. As that number grows and the complexity becomes harder and harder to handle, a switch to either of the presented heuristics can be made. With this in mind, the compromise between complexity and the best solution is reduced and a better algorithm is presented.

# Use Cases

It is very difficult to find parking spots when traveling around the city and the solution to this problem is the basis for a mobile app that would help people select the best trip around a city and find parking lots near their desired locations. However, this project has a serious limitation: it is currently impossible to have all of the data needed to have a correct output. One of the most important data entries that is not easy to obtain is the park availability, since there is no service that provides that information. The solution to that problem is to allow the users to cooperate and share data with everyone, meaning that in parks that lack that information, the users would provide status on the park upon reaching there.

## 4.1 Actors

The applicability of this project is based on user cooperation and shared information, but an admin is also needed.

1. **User** - The application is centralized on the user. It will be him who will use the application to travel around the city and choose parameters to have a customized trip. It will be also the user who'll make it possible to have relatively updated information concerning a park's current capacity.
2. **Admin** - The admin account has the ability to manage available parking spots and edit the map for unavailable streets, for example, adding or removing information to/from the map.

## 4.2 Project Functionalities

This project will run in a terminal interface where the user can input the data previously stated. The user will choose the starting and destination points as well as the points of interest before reaching the destination. He will also type in the terminal the importance factors of the walking distance and parking price to better serve the user when choosing a park. Since the parks are affected by availability, it would be interesting to simulate users concurrency, trying to access parks at the same time. After the result is calculated, an animation will be displayed on a graph, showing the trajectory that the user should take.

# Main Considerations and Preliminary Analysis

The problem at hand, which led to ***MyJourney!***, was identified as being interesting, challenging and quite pertinent. The purpose of this report was to research and identify the problem and the algorithms needed to reach a solution.

In order to better understand and quickly approach the problem, the group performed an analysis of a real world application of graphs in mapping systems, *OpenStreetMap*. This helped understand how streets and parking lots are represented in a graph, resulting in a better formalization of the input data needed. Next followed the discussion of the various steps to solve the problem and the identification of a 'Travelling Salesman'-like problem.

As such, the solution was split in three steps (Section 3.1):

1. Finding the best parking spot, using Dijkstra's shortest path algorithm and filtering down the candidates (Subsection 3.1.2).
2. Finding the shortest path to such candidate/s, using the A* Algorithm (Subsection 3.1.3) using an heuristic as defined in subsection 3.1.3.
3. Solving the 'Travelling Salesman'-like problem in finding the or an approximation of the best path to travel through all points of interest before reaching the destination (Section 3.2) which, by itself, also requires multiple steps:
   (a) Adapting the problem to a Travelling Salesman problem as explained in section 3.2 and in Figure 3.1.
   (b) Solving this adaptation either by using the Held-Karp Algorithm (Subsection 3.2.1) or by using either heuristics referenced in subsection 3.2.2, *Nearest Neighbour* and *Minimum Spanning Trees.*

One of the main goals of this solution was to balance time complexity and accuracy, as it represents a real-life application that needs good and relatively fast results 4.

Nevertheless, some issues may need to be addressed such as the work-around used to circumvent the impossibility of using directed graphs with the *Minimum Spanning Trees* heuristic (subsection 3.2.2) which can worsen its accuracy. In addition, finding the perfect compromise between complexity and accuracy (Subsection 3.2.3), in choosing which algorithms to use in the Travelling Salesman Adaptation, can become hard as the line between what is considered a low or high amount of points of interest can be blurry.

Despite these issues, the break down of the problem assigned to the group allowed a more organized, thought-out and coordinated solution which will surely reduce the complexity and time needed to fulfill the goals of the project.

# Main Functionalities And Scenarios Implemented

All defined functionalities and all algorithms (with the exception of Held-Karp) were effectively implemented, tested and ready for application use.

The main program flow works as intended, and the interactive view created to output the result makes it easier to interact and verify its validity.

For each point of interest chosen, the chosen parking spot is displayed, as well as the other possibilities it had - making it easier to verify that the best park was in fact chosen. The application allows for the preferences of the user to be changed in regards to the time to park and the max radius to search, and those restrictions can be very well noticed in the interactive view.

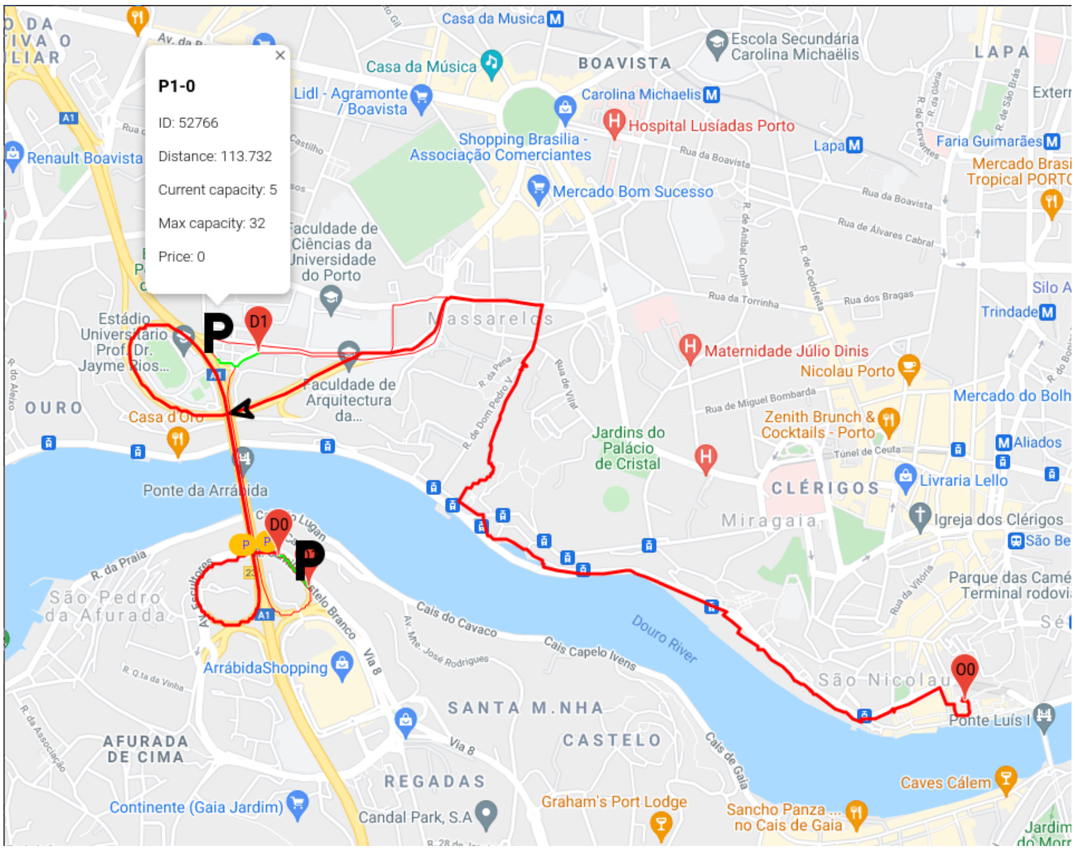
The validity of the SSP solution can also be effectively verified with the aid of the interactive view.

## 6.1 Basic Features

The program runs initially on a terminal interface where the user can login. There are two types of users, an admin, who can analyse connectivity and remove parks from the map, and a normal user, who is able to generate journeys and provide feedback about some parks information like current capacity. In a real life situation this is crucial since most parks don't have the technology necessary to share its information, and we need a way to maintain the algorithm working as well as possible when choosing the park.

## 6.2 Extra features

We decided to implement the visual part using the ***Google Maps API***. With it, it is possible to interact very well with the map, making this experience more interactive and user friendly. Using some features this API provides, we were able to show the nodes chosen by the user, the parks analysed and chosen by the algorithm and park information like prices and capacities. Another feature that really helps the user to use the software is to click in the map and receive information about latitude and longitude. Using the information, we can introduce those values in the program, where it will find a nearby point. The user can also choose points of interest and nodes using their id.



P1-0

ID: 52766

Distance: 113.732

Current capacity: 5

Max capacity: 32

Price: 0

Figure 6.1: Interactive View of the Application Result

# Data Structures Used

## 7.1 Basic Structures

1. Disjoint Set using path compression. Used for calculating minimum spanning trees using kruskal's algorithm.
2. Position. Stores a position in either latitude [-90, 90] and longitude [-180, 180] or in X,Y Cartesian grid. Used for calculating distances
3. Priority Queue. Used in Dijkstra's algorithm.
4. Map. Used to store nodes in graphs.

## 7.2 Complex Structures

### 7.2.1 Graph

The graph structure stores and controls a map of ids to pointers of nodes, representing a graph. The graph structure can be used to add nodes, add edges and find nodes. An adjacency list is used.

#### Node

A node structure contains multiple variables used by different algorithms. Each node has an ID which is its index in the graph map. Every node has a list of outgoing and incoming edge pointers. The 'path' parameter used in path finding algorithms (representing the previous node in a path) is stored as a node pointer.

#### Edge

An edge structure represents an edge, storing the origin and destination nodes and its cost. An edge can also know its reversed edge, stored as a pointer.

#### Undirected Graph

An undirected graph extends the graph class. It overrides the function to add edges making it so whenever an edge is added, a reversed edge is added to the outgoing list of the destination node. A constructor is also used to create a graph of nodes where the edges are the euclidean distances between them. It is used by the Kruskal Minimum Spanning Trees algorithm.

### 7.2.2 NodeInfo

The NodeInfo structure stores the info on a node of a street map. It stores the node's type (park or normal node) and, if it is a park, its current and max capacity and its price function.

# Algorithms Implemented

## 8.1 Dijkstra Parks

This algorithm, as previously explained, was adapted to the problem at hand in order to search for parks within a given radius from the destination. The implementation pseudo-code was already exposed previously as well. We later realized that we needed to run this on some kind of bi-directed graph, since walking can follow both ways. To do this, we added a **walking** vector that contained both incoming and outgoing edges.

## 8.2 Dijkstra Path Finding

This algorithm was added onto the project in order to have some algorithm to compare A* to. Its pseudo-code is similar to the A* one, but without the heuristic calculation.

## 8.3 A*

A*'s implementation, as well as its pseudo-code, was already thoroughly explained here.

## 8.4 Kruskal Minimum Spanning Trees

Kruskal's implementation was also adapted, as previously explained. It generates a minimum spanning tree with the given points of interest. Its pseudo-code is also thoroughly analysed and exposed here;

## 8.5 Nearest Neighbour

Just as the MST algorithm, this implementation of the Nearest Neighbour generates an ordered list of the user's points of interest. Its pseudo-code is also exposed previously.

## 8.6 Tarjan's Algorithm

This algorithm was already explained previously and is yet again referenced in chapter 10.

## 8.7 Brute-Force Traveling Salesman

This algorithm was adapted to the problem and is different than the usual implementation. It only generates all permutations of the points of interest and then calculates the total distance counting the origin and destination nodes.

**Algorithm 9** TSP using Brute-Force

```
1:  function BRUTEFORCE($poi, origin, destination$)
2:      if $poi$ is $empty$ then return $\{origin, destination\}$
3:      end if
4:      $shortDistance \leftarrow \infty$
5:      $shortestPath \leftarrow \{\}$
6:      $sort(poi)$ //Sorting the points of interest is necessary to generate all permutations
    lexicographically
7:      repeat
8:          $distance \leftarrow 0$
9:          $first \leftarrow origin$
10:         $second \leftarrow firstAt(poi)$
11:         $distance \leftarrow distance + dist(first, second)$
12:         for i in range $(0, size(poi) - 1)$ do
13:             $first \leftarrow poi[i]$
14:             $second \leftarrow poi[i + 1]$
15:             $distance \leftarrow distance + dist(first, second)$
16:         end for
17:         $first \leftarrow lastAt(poi)$
18:         $second \leftarrow destination$
19:         $distance \leftarrow distance + dist(first, second)$
20:         if $distance \leqslant shortDistance$ then
21:             $shortDistance \leftarrow distance$
22:             $shortestPath \leftarrow copy(poi)$
23:         end if
24:     until $getNextPermutation(poi) = false$
25:     $insertBegin(shortestPath, origin)$
26:     $push(shortestPath, destination)$
27:     return $shortestPath$
28: end function
```

# Complexity Analysis of the Algorithms Implemented

## 9.1 Path Finding

### 9.1.1 Theoretical Analysis

Dijkstra's algorithm, implemented with a priority queue, runs in time $\Theta((|E|+|V|)log(|V|))$ (exact bound) and has a space complexity of $O(|V|)$, while A* has a worst-case time complexity of $O(|E|)$ and a worst-case space complexity of $O(|V|)$.

### 9.1.2 Empirical Analysis

The results in the charts in 9.3 and 9.4 are generated in increments of 8 nodes, from 8 to 1024 nodes. In each increment, 256 samples in the form of randomly generated graphs are generated. Each path finding algorithm is given the same origin and destination node and each node has 4 outgoing edges.

The chart in 9.3 is the result of the sums of the time executions in each sample of each increment, then divided by the number of samples (256), resulting in a mean time per increment.

The chart in 9.4 is the result of recording every result of each sample in each increment and getting the median of the results of each increment.

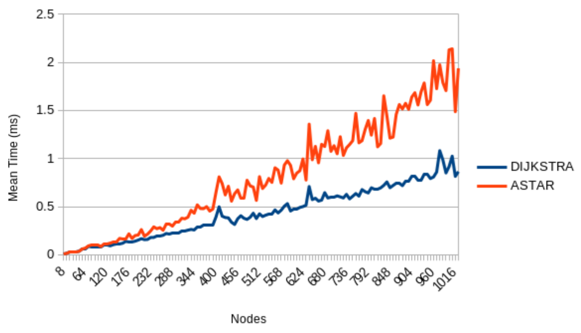


Figure 9.1: Mean of times for path finding. Blue stands for Dijkstra, red stands for A*

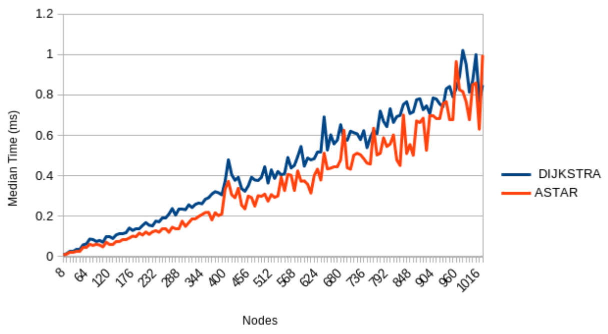


Figure 9.2: Median of times for path finding. Blue stands for Dijkstra, red stands for A*

From these results, it is possible to see that our implementation of Dijkstra is faster on average against our A*. This might be because of the slow heuristic used which is calculating distance from latitude and longitude since it uses slow functions like $sin()$ and $cos()$ and floating point arithmetic. However the median results for A*, while quite similar, are faster than Dijkstra's.

Dijkstra's average times are closer to its median times than A*'s results.

Creating a trend line for Dijkstra with the equation of $\frac{(|E|+|V|)log(|V|)}{3000}$ for the mean and $\frac{(|E|+|V|)log(|V|)}{6000}$ for the median, and a trend line for A* with $\frac{|E|}{2200}$ for the mean and $\frac{|E|}{4400}$ for the median it becomes clear that the algorithms implemented follow the theoretical analysis.

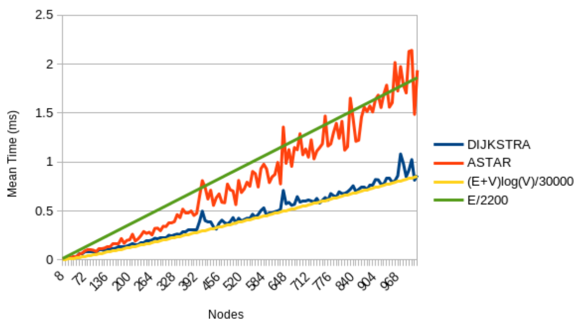


Figure 9.3: Mean of times for path finding. Blue stands for Dijkstra, red stands for A*

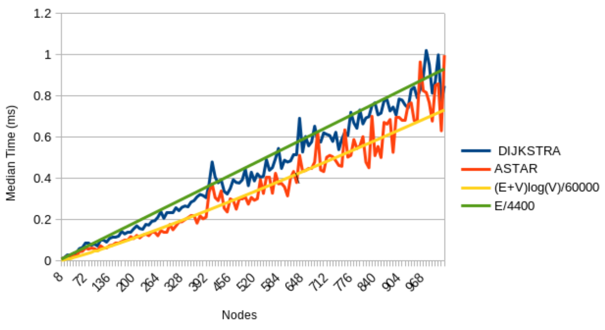


Figure 9.4: Median of times for path finding. Blue stands for Dijkstra, red stands for A*

## 9.2 Traveling Salesman

### 9.2.1 Theoretical analysis

The brute force algorithm for solving Traveling Salesman problems has a time complexity of $O(n!)$ and a spacial complexity of $O(1)$.

Kruskal's algorithm (visiting afterwards with a depth first search) has a time complexity of $O(|E|log(|E|) + |V| + |E|) = O(|E|log(|E|))$ normally, however in this case, since $|E| = |V| \times |V|$ because each node has an edge to every other node, we have a complexity of $O(|V^2|log(|V^2|)) = O(|V|^2 \times 2 \times log(|V|)) = O(|V|^2 log(|V|))$. It has spacial complexity of $O(|V|)$ for disjoint sets and $O(|V|)$ for the stack used in the depth first search, resulting in a complexity of $O(2 \times |V|) = O(|V|)$.

The nearest neighbour algorithm has a time complexity of $O(|V|^2)$ and a spacial complexity of $O(1)$.

### 9.2.2 Empirical analysis

The results in the following charts are generated in increments of 1 node, from 2 to 256 nodes. In each increment, 32 samples in the form of randomly generated graphs are generated. Each TSP algorithm is given the same origin and destination node. All algorithms also execute operations on the same set of points of interest.

The chart in 9.5 and 9.7 is the result of the sums of the time executions in each sample of each increment (up until 10 points of interest for 9.5), then divided by the number of samples (32), resulting in a mean time per increment.

The chart in 9.6 and 9.8 is the result of recording every result of each sample in each increment (up until 10 points of interest for 9.6) and getting the median of the results of each increment.

No further increments were assessed for the brute force algorithm as the time required to compute the results started getting too costly. In other attempts, at 11 points of interest, brute force got up to the order of 86 seconds and at 12 points of interest brute force got up to 1 million milliseconds which is in the order of 17 minutes.

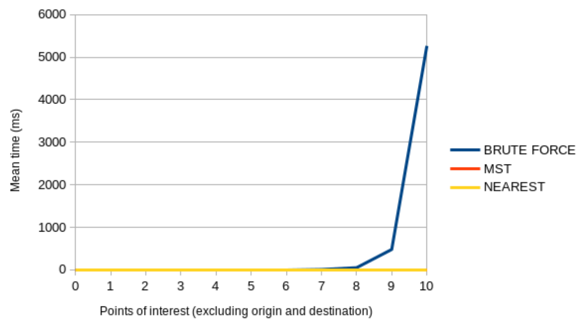


Figure 9.5: Mean of times for Traveling Salesman. Blue is Brute Force, red is Kruskal, yellow is Nearest Neighbour

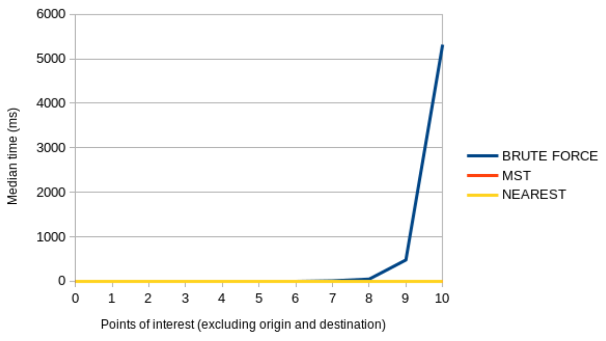


Figure 9.6: Median of times for Traveling Salesman. Blue is Brute Force, red is Kruskal, yellow is Nearest Neighbour

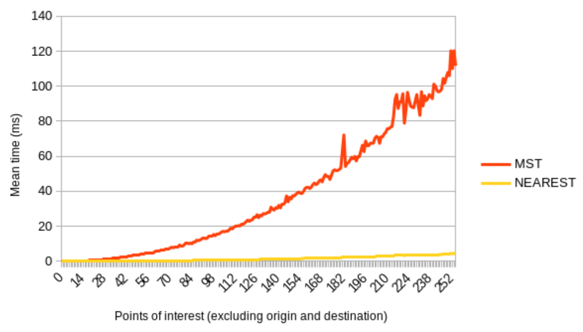


Figure 9.7: Mean of times for Traveling Salesman. Red is Kruskal, yellow is Nearest Neighbour

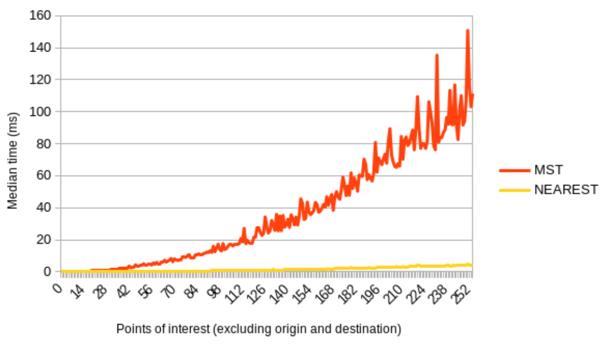


Figure 9.8: Median of times for Traveling Salesman. Red is Kruskal, yellow is Nearest Neighbour

From a first analysis, unlike the path finding algorithms, the median is very similar to the mean times in every chart. This could be because of the low sample size, however. Because of this, the median times will not be analysed.

From the results of 9.5 and 9.6 we can conclude that the brute force algorithm starts becoming too costly for any real use at around 8 or 9 points of interest. We can also conclude that the brute force algorithm is orders of magnitude more complex than either Kruskal's algorithm or Nearest neighbour. It is also possible to conclude that not only is our implementation of Kruskal's algorithm slower than the Nearest Neighbour algorithm it also grows faster in time.

Creating a trend line for the Brute Force algorithm with the equation of $\frac{|N!|}{700}$ with N being the number of points of interest, a trend line for Kruskal with $\frac{|V|^2 log(|V|)}{17000}$ and a trend line for Nearest Neighbour with $\frac{|V|^2}{15000}$ it becomes clear that the algorithms implemented follow the theoretical analysis.

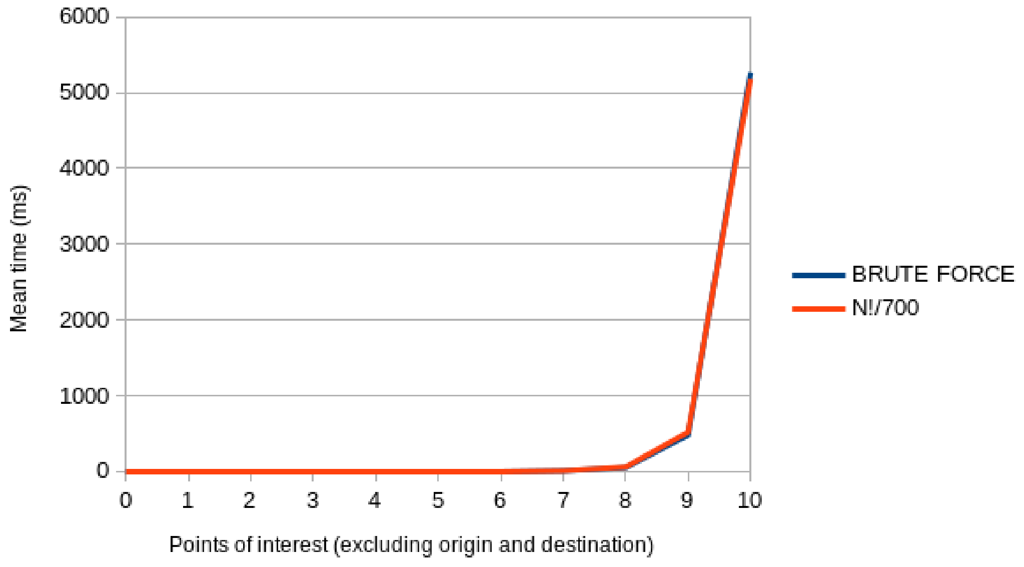


Figure 9.9: Mean of times for Traveling Salesman. Blue is brute force, red is the trend line.

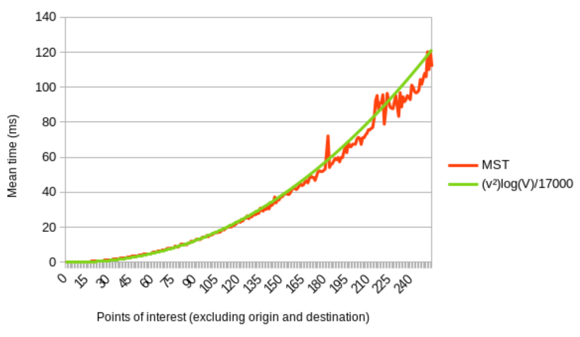


Figure 9.10: Mean of times for Traveling Salesman. Red is Kruskal, yellow is the trend line.

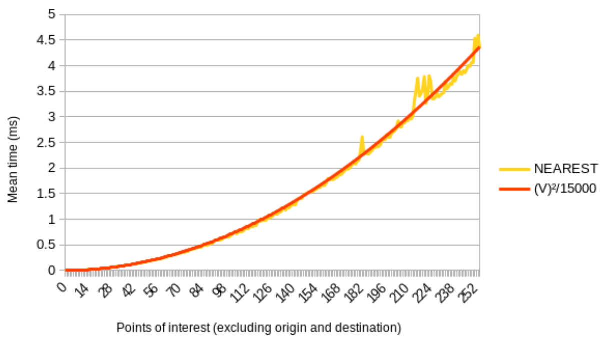


Figure 9.11: Mean of times for Traveling Salesman. Yellow is Nearest Neighbour, red is the trend line.

The chart in 9.12 is the result of the sums of the percetages of the optimal solution (brute force solution) in each sample of each increment (up until 10 points of interest), then divided by the number of samples (32), resulting in a mean percentage per increment.

The chart in 9.13 is the result of the sums of the distances of the solutions of each sample in each increment (starting from 11 points of interest), then divided by the number of samples (32), resulting in a mean distance travelled per increment.

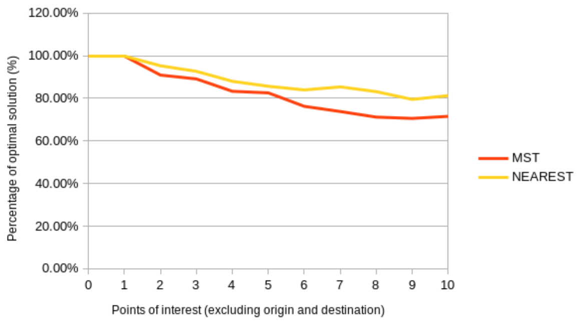


Figure 9.12: Mean percentage of the optimal solution per algorithm. Red is Kruskal, yellow is Nearest Neighbour

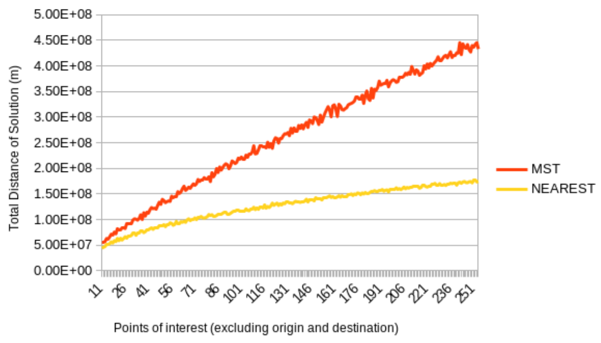


Figure 9.13: Mean Distance travelled by the solutions of each algorithm. Red is Kruskal, yellow is Nearest Neighbour

From this we can conclude that indeed, at least for the first 10 points of interest, the Minimum Spanning trees results in 15% to 20% of the optimal solution, as stated in chapter 3.2.2. However we can also conclude that, in average, the Nearest Neighbour algorithm has better results than the MST approach.

In conclusion, we can use the brute force algorithm up to 7 or 8 points of interest, resulting in the optimal solution quickly. After that point, a compromise can be made by using the Nearest Neighbour algorithm instead, resulting in much quicker times, but worse results (around 80% of the optimal solution following the chart 9.12)

# Analysis of the graph connectivity

Using the previously mentioned Tarjan's Algorithm, the graph connectivity can be verified in the application by an administrator.

Whenever this option is chosen, information about the graph's connectivity is shown on the screen: if it is strongly connected or not, how many strongly connected components the graph does have and how many nodes, on average, does each strongly connected component have.

This analysis is pertinent to assure that the algorithm runs properly. The nodes must belong to the same SCC to guarantee that it finds a solution, and verifying this condition on the input map will make sure that the program's execution is the one desired.

# Conclusion

As mentioned before, most algorithms were implemented and are working as expected. It's clear, however, that their performance can be tweaked and improved.

For example, A* implementation generates good results, but it is clear, sometimes, that the shortest path was not found - the solution led to a greater distance cost. These problems might arise from a variety of factors, such as the distance functions not being as precise as they should, the map not being perfectly matched to the reality or even other unknown code bugs that might not have been found.

The theoretical analysis of the TSP problems must also be scrutinized: MST ended up performing worse than the Nearest Neighbour, which was something we did not anticipate. The conclusion, however, was clear and made sense when really diving deep into the reasoning of why it happened.

Overall, however, the project is working as expected and the proposed problem is being solved relatively well. As it is difficult to really measure the performance of the overall solution, as there's no possible comparison, the calculated solutions seem reasonable and appliable to a real-life application.

## 11.1 Final Remarks

While the formalization of the problem appeared to be quite challenging at times, it helped immensely on the actual implementation, as the problem was well thought-out and properly organized in advance.

The actual project concept helped us get in touch with these algorithms and problems in a deeper manner, which was pretty useful to really understand the curricular unit's theoretic conceptions.

# Contributions

All members of the group worked together to come up with this solution and its report. The effort was distributed evenly and, as such, each member has a 33.3% contribution rate.

All members were equally involved in the formalization of the problem and definition of the input data, output data, restrictions and objective functions.

André Moreira took a deeper approach to the TSP problem, formalizing its heuristics and possible algorithms. He also was responsible for the Dijkstra implementation and the view interface model with Google Maps API.

Nuno Alves formalized the Dijkstra solution and its intricacies, as well as the use cases and main considerations to take from this project. He was also responsible for the Kruskal's and Nearest Neighbour implementation, as well as the brute force algorithm for solving TSP and the empirical performance tests.

Nuno Costa approached the graph connectivity problem and the A* adaptation for the path problem. He also was responsible for the implementation of these algorithms, as well as the application model for creating users and choosing the journey parameters and its generation.

It must be noted, however, that even though all members were involved in distinct parts in a deeper manner, the cooperation in all aspects and chapters of this project existed and was maintained throughout.

# Bibliography

[1] *Tarjan's Strongly Connected Component (SCC) Algorithm (UPDATED) | Graph Theory*. YouTube, Apr. 2020. URL: https://www.youtube.com/watch?v=wUgWX0nc4NY&amp;t=55s (page 8).

[2] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136 (page 11).

[3] Michaël Carlos Gonçalves Adaixo. *Influence Map-Based Pathfinding Algorithms in-Video Games*. URL: https://ubibliorum.ubi.pt/bitstream/10400.6/5517/1/3443_6881.pdf (page 11).

[4] Vladimir Agafonkin. *Fast geodesic approximations with Cheap Ruler*. Aug. 2017. URL: https://blog.mapbox.com/fast-geodesic-approximations-with-cheap-ruler-106f229ad016 (page 12).

[5] Steven S. Skiena. *The Algorithm Design Manual*. London: Springer, 2008. ISBN: 9781848000704 1848000707 9781848000698 1848000693. DOI: 10.1007/978-1-84800-070-4 (pages 14, 16).

[6] Susan N. Twohig and Samuel O. Aletan. "The Traveling-Salesman Problem (Abstract)". In: *Proceedings of the 1990 ACM Annual Conference on Cooperation*. CSC '90. Washington, D.C., USA: Association for Computing Machinery, 1990, p. 437. ISBN: 0897913485. DOI: 10.1145/100348.100468. URL: https://doi.org/10.1145/100348.100468 (page 14).

[7] Kazuro KIMURA, Shinya HIGA, Masao Okita, and Fumihiko Ino. "Accelerating the Held-Karp Algorithm for the Symmetric Traveling Salesman Problem". In: *IEICE Transactions on Information and Systems* E102.D (Dec. 2019), pp. 2329–2340. DOI: 10.1587/transinf.2019PAP0008 (page 14).