E-Stafetas - merchandise delivery by electrical vehicles

Conceção e Análise de Algoritmos - Part I

Class 3 group 2

up201906086@edu.fe.up.pt Marcelo Henriques Couto
up201807481@edu.fe.up.pt Afonso Marques Cabral de Carvalho
up201705110@edu.fe.up.pt João Luis Cardoso Rodrigo

16 of April, 2021

Contents

1 Executive Summary **3**

2 Problem Description **4**
- 2.1 Introduction . . . 4
- 2.2 Shortest Route Problem . . . 4
- 2.3 Autonomy Problem . . . 5
- 2.4 Connectivity problem . . . 5
- 2.5 Orders Distribution . . . 5

3 Problem Formalization **6**
- 3.1 Input Data . . . 6
- 3.2 Output Data . . . 7
- 3.3 Restrictions . . . 8
  - 3.3.1 Input Restrictions . . . 8
  - 3.3.2 Output Restrictions . . . 9
- 3.4 Objective Function . . . 9

4 Algorithms **11**
- 4.1 Shortest Path Between Two Nodes . . . 11
  - 4.1.1 Dijkstra's Algorithm . . . 11
  - 4.1.2 Bellman-Ford Algorithm . . . 13
  - 4.1.3 Bidirectional Dijkstra . . . 15
  - 4.1.4 A* Algorithms . . . 18
- 4.2 Shortest Path Between All Nodes . . . 19
  - 4.2.1 Floyd-Warshall . . . 19
  - 4.2.2 Dijkstra's Algorithm . . . 21
- 4.3 Connectivity . . . 22
  - 4.3.1 Brute Force with Floyd-Warshall . . . 22
  - 4.3.2 Kosaraju's algorithm . . . 22

4.3.3 Tarjan's Algorithm 25
4.4 Clustering Algorithms 28
4.4.1 Agglomerative Hierarchical Clustering 28
4.5 Extras 29

5 Prospective Solution **30**
5.1 Pre-Processing of Input Data 30
5.2 Problem Identification 30
5.3 Solutions 31
5.3.1 Problem I - Itineraries/Routes Calculation 31
5.3.2 Problem II - Autonomy/Range 31
5.3.3 Problem III - Connectivity 31
5.3.4 Problem IV - Orders Distribution 32
5.3.5 General 32

6 Functionalities Implemented **33**

7 Conclusion **35**
7.1 Tasks allocation 35

8 Bibliography **36**

Chapter 1

Executive Summary

In this report, we will analyse a the problem given to us in the context of the practical work of Conception and Analysis of Algorithms. After **presenting and decomposing the problem**, a **formalization** will be done, followed by presentation of the **possible algorithms** to implement int the solution. Finally, a **prospective solution** will be given, compiling the chosen algorithms and a strategy. Lastly, a list of the **functionalities** the interface of the program to be built in phase two is presented and a **conclusion** is given.

Chapter 2

Problem Description

2.1 Introduction

A home delivery company, motivated by the rise in online sales and the increasing necessity to preserve our planet, decided to invest in electrical vehicles to perform its deliveries. For the idea to work, a system to manage the routs of the vehicles must be implemented. The system must have more factors into account than usual, given these vehicles' more restricted range and the difference in the fuelling process.

Each vehicle from the company's fleet has to perform multiple deliveries daily. Each delivery can be summed up to three steps:

- Picking up the product(s) from the pick up point
- (optional) Recharge the vehicle in a recharge or in the enterprise's garage
- Delivery of those same product(s) on the client's address

The objective is to calculate the itinerary for the day for all the vehicles given a set of orders.

2.2 Shortest Route Problem

In order to optimize the use of the company's vehicles and shorten their itineraries, the paths calculated between pick up and delivery points (and any other points points) must be as short and fast as possible.

2.3 Autonomy Problem

Given the reduced distances the electrical vehicles are capable of covering with 'full tank' in comparison to a traditional vehicle, the autonomy is an important factor when calculating the optimal itinerary for the vehicle. There are two strategies available for when a vehicle does not have enough fuel to perform a delivery:

- The route must go through the company's garage so that the vehicle can recharge there
- The route must go through one of many recharge points scattered throughout the map

In a way, the first strategy is simply a special case of the second one, where the recharge point is unique and happens to be in the garage's location.

2.4 Connectivity problem

Another problem which needs to be tackled is the one of the unexpected events that might change the availability of certain roads and paths. At any given time, there may be accidents on the vehicle accidents or road works that block the passage on certain spots that could be part of the optimal path. This demands daily checks, re-calculation of the itinerary and may even, in rare cases, deem the delivery impossible for the moment.

2.5 Orders Distribution

Due to the reduced amount of vehicles available for the delivery, orders must be intelligently distributed through the drivers to make the usage of the vehicles the most efficient.

Chapter 3

Problem Formalization

3.1 Input Data

Notes

- The geographical information will be loaded in the form of a map, obtained in OpenStreetMap and converted to a graph
- The words 'street', 'road' and 'way' are used interchangeably
- The words 'node' and 'vertex' ('nodes' and 'vertices') are used interchangeably
- The words 'range' and 'autonomy' are used interchangeably The words 'rout' and 'itinerary' are used interchangeably

Graph

G = (N, E) - weighted directed (roads may be one way only) graph that represents the map. It is composed by Nodes and Edges

- N - set of Nodes (a node represents an *interest point* or simply a point in the road network) (N(i) is the ith element). For each node:
  - Adj - edges whose origin is N(i)
  - lat - the latitude of the point it represents in the map
  - long - the longitude of the point it represents in the map
- A - set of Edges (an edge represents a way/street/road) (E(i) is the ith element). For each edge:

  - w - weight (represents the length of the way) (measured in meters)
  - dest - origin node of the edge
  - orig - destination node of the edge

Interest Points (special nodes)

- base - base/garage of the company
- rp - recharge points
- pup - pick up point
- dp - delivery point

Vehicles

V - set of vehicles that make the fleet of the company (Ve(i) is the ith element). For each vehicle:

- range - current range of the vehicle (in kilometers)

Orders

O - set of orders for the day (O(i) is the ith element). For each order:

- pup - pick up point
- dp - delivery point

3.2 Output Data

Each vehicle is assigned a *path* that represents the best sequence of nodes for the orders *Ov* he's been assigned to. The path and orders are distributed taking to account the optimization of the distance travelled and the range of each vehicle at any given time. For each Vehicle V(i):

- path - sequence of nodes that represents the path to be taken by a vehicle on a day (path(i) is the ith element)
- Ov - set of orders assigned to the vehicle (OrV(i) is the ith element)

3.3 Restrictions

3.3.1 Input Restrictions

General

- The sets' indexes are implicitly limited by their sizes and 0
  Example: $0 \leq i < |N|$

Graph

- Node
  - $|N| > 0$
  - $\forall n \in N, Adj(n) \subseteq E$ (Adj is the set of edges originated in N(i))
  - $\forall n \in N, 0° \leq lat \leq 360°$
  - $\forall n \in N, 0° \leq long \leq 360°$
- Edge
  - $|E| > 0$
  - $\forall e \in E, w > 0$ (the length of a way or road must be always bigger than 0)
  - $\forall e \in E, orig \in N$
  - $\forall e \in E, dest \in N$
- $base \in N$
- $rp \in N$
- $pup \in N$
- $dp \in N$
- $\exists! n \in N, n = base$

Vehicles

- $|V| > 0$
- $\forall v \in V, range > 0$

Orders

- $|O| \geq 0$
- $\forall o \in O, pup \in N$
- $\forall o \in O, dp \in N$

3.3.2 Output Restrictions

Path

- $|path| > 0$
- $path \subseteq N$
- $\forall p \in path, p \in N$

Vehicle Orders

- $|Ov| \geq 0$
- $Ov \subseteq O$
- $\forall o \in Ov, o \in O$

3.4 Objective Function

The company's goal for this system is to optimize the use of their vehicles. As we will not consider different speeds of travel for the vehicles, the time to fulfill a order behaves in parallel to the distance of its path. In our interpretation of the problem, optimizing the use of the vehicles would mean to find the shortest paths between two interest points in a rout, as well as minimizing the distance of the rout itself. In an optimal solution, the maximum distance travelled by a vehicle in a day would also be minimized. As such, there must be three functions to optimize (ones depend on the others). Being f the function that represents the most distance travelled by a vehicle; g the sum of the distances between the points in an itinerary; h the function that represents the distance between two interest points: the objective is to minimize them:

$$f = \max(D)$$

- D being the set of distances travelled by each vehicle in a day

$$g = \max(sum(d))$$

$$h = \max(d)$$

- d_1 being the distance from one interest point to another

Chapter 4

Algorithms

In this chapter, we will list some important algorithms which may be used in the solution of each subproblem.

4.1 Shortest Path Between Two Nodes

Algorithms for the calculation of the Shortest Path between two points are very important for the solution of this problem. They will be used to calculate the shortest path between multiple points when calculating a vehicle's path, such as the shortest path between the garage and a pick up point. To do this, there are many algorithms at our disposal. We will evaluate some of them and make a decision on which is(are) best for the situation at hand.

4.1.1 Dijkstra's Algorithm

Description

This algorithm was conceived by Edsger W. Dijkstra's in 1956 and is used to calculate shortest paths in directed or undirected graphs, so long there are no edges with negative weight. Upon utilization of this algorithm, a tree holding the shortest paths from the origin node to all others is formed.

Extra Data, Data Structures and Algorithms Required

- Minimum distance to the origin node from each of the graph's nodes (dist)
- Node that comes before it in its path (path)

- Priority queue to hold the nodes to be processed next
- Decrease-Key function, to maintain the nodes with shortest distance in the top of the queue

Algorithm 1 Pseudo-Code for Dijkstra's Algorithm

```
0: function DIJKSTRA(G(N, E), s)
1: for all n ∈ N do
2:    dist(n) ← ∞
3:    path(n) ← nil
4: end for
5: dist(s) ← 0
6: Q ← ∅ {Q is the priority queue (min)}
7: Insert(Q, (s, 0) {inserts s with key 0}
8: while Q ≠ ∅ do
9:    n ← top(Q) {minimum in Q, which is the top}
10:   pop(Q) {extracts min of Q (pops)}
11:   for all (n, n2) ∈ Adj(n) do
12:      if dist(n2) > dist(n) + weight(n, n2) then
13:         dist(n2) ← dist(n) + weight(n, n2)
14:         path(n2) ← n
15:         if n2 ∈ Q then
16:            Insert(Q, (n2, dist(n2)))
17:         else
18:            Decrease − Key(Q, (n2, dist(n2))) {function to decrease the key in Q}
19:         end if
20:      end if
21:   end for
22: end while
22: end function=0
```

Analysis

The first part of the algorithm's goal is to prepare the data (lines 1-6): paths are set to null, distances to infinity, the priority queue is initialized and the origin node is inserted. Then, a breadth-first search is performed; a check is made to every node found in order to understand if its path can be shortened by the use of the edge in analysis. After that, if the node is not already in the queue, it must be inserted, to

be eventually processed. If it is, its key must be reduced. The items of the priority queue are ordered by their key. This system is used so that the nodes which are closest to the origin are processed first. This procedure ensures that the distance of already processed nodes remains intact, boosting the algorithm's efficiency and making this a greedy algorithm.

Efficiency

- **Temporal Complexity** - $O((|N| + |E|) \times \log(|N|)$, where E is the set of edges and N the set of nodes of the graph. Decrease-Key's time efficiency is $O(|E| \times log(|N|)$. It can be $O(1)$ if Fibonacci Heaps are used instead of regular Priority Queues, making the whole algorithm $O(|N| \times log(|N|)$
- **Spatial Complexity** - Depends on the implementation, but usually $O(|N|)$

Usability

This algorithm is one to consider using in the problem, but there may be slightly more efficient ones.

4.1.2 Bellman-Ford Algorithm

Description

The Bellman-Ford Algorithm is another algorithm used to calculate the shortest path between nodes in a graph. It was first proposed by Alfonso Shimbel in 1955 but ended up being named after Richard Bellman and Lester Ford, who later officially published it independently. This algorithm is useful to calculate shortest paths in graphs with negative weight edges and to detect negative edge cycles in them.

Extra Data, Data Structures and Algorithms Required

- Minimum distance to the origin node from each of the graph's nodes (dist)
- Node that comes before it in its path (path)

Algorithm 2 Pseudo-Code for Bellman-Ford

```
 0: function BELLMAN-FORD(G(N, E), s)
 1: for all n ∈ N do
 2:    dist(n) ← ∞
 3:    path(n) ← nil
 4: end for
 5: dist(s) ← 0
 6: for i = 1 to |N| − 1 do
 7:    for all (n1, n2) ∈ E do
 8:       if dist(n2) > dist(n1) + weight(n1, n2) then
 9:          dist(n2) ← dist(n1) + weight(n1, n2)
10:          path(n2) ← n1
11:       end if
12:    end for
13: end for
14: for all (n1, n2) ∈ E do
15:    if dist(n1) + weight(n1, n2) < dist(n2) then
16:       print fail: negative cycle
17:    end if
18: end for
18: end function=0
```

Analysis

In this algorithm, the first step is to prepare the graph (lines 1-4). Then, all edges are analyzed $|N| - 1$ times(lines 6-7), checking if they can be used to shorten the path from s to the destination node of the edge(line 8). This process is repeated this amount of times because it is the maximum length in edges for a path. In the end, a check is made, to see if the algorithm worked. If it did not, the cycle as negative weight loops (lines 14-18). This algorithm is considered to implement dynamic programming.

Efficiency

- **Temporal Complexity** - $O(|N| \times |E|)$, where E is the set of edges and N the set of nodes of the graph
- **Spatial Complexity** - Depends on the implementation, but in this case $O(1)$

Usability

This algorithm poses little interest to this problem given that our edges' weights represent the length of a road, which would never be negative.

4.1.3 Bidirectional Dijkstra

Description

Ira Pohl was the first one to design and implement a bidirectional heuristic search algorithm in 1971. Bidirectional Dijkstra's that implements the idea of a bidirectional search into Dijkstra's algorithm. It finds a shortest path from an initial node to a goal node in a directed or undirected graph. It runs two simultaneous searches: one forward from the initial state, and one backward from the goal, alternating between the two. The stopping criteria is one that must be very well implemented to guarantee the shortest path. Compared to normal Dijkstra's there's a speedup of 2x.

Extra Data, Data Structures and Algorithms Required

- Minimum distances to the origin and goal nodes from each of the graph's nodes (dist)
- Nodes that come before each node in both forward and backward paths (not implemented in pseudocode)
- Two priority queues to hold the nodes to be processed next in each direction
- Way of search (way) (1 if it was encountered by forward search first, 2 if it was encountered by backward search first, 0 default)
- Decrease-Key function, to maintain the nodes with shortest distance in the top of the queue

Algorithm 3 Pseudo-Code for Bidirectional Dijkstra's Algorithm

0: **function** BIDIRECTIONALDIJKSTRA($G(N, E), s, p$) {s is the starting node and p the ending node}
1: **for all** $n \in N$ **do**
2: $dist(n) \leftarrow \infty$
3: $distR(n) \leftarrow \infty$ {Distance backward}
4: $way(n) \leftarrow 0$
5: **end for**
6: $bestDist \leftarrow$ inf
7: $dist(s) \leftarrow 0$, $distR(p) \leftarrow 0$
8: $way(s) \leftarrow 1$, $way(p) \leftarrow 2$
9: $Qs \leftarrow \emptyset$, $Qp \leftarrow \emptyset$ {Qs and Qp are the priority queues for the forward and backward search side respectively}
10: $Insert(Qs, (s, 0)$, $Insert(Qp, (p, 0)$ {inserts s and p with key 0}
11: **while** $Q \neq \emptyset$ **do**
12: $n \leftarrow extractMin(Qs)$, $v \leftarrow extractMin(Qp)$ {extracts the min from the priority queue}
13: **if** $dist(n) + distR(v) \geq bestDist$ **then**
14: **return** $bestDist$
15: **end if**
16: **for all** $(n, n2) \in Adj(n)$ **do**
17: **if** $way(n2) = 2$ **and** $bestDist > distR(n2) + weight(n, n2) + dist(n)$ **then**
18: $bestDist \leftarrow distR(n2) + weight(n, n2) + dist(n)$
19: **else**
20: $way(n2) \leftarrow 1$
21: $MiniDijkstraStep(Qs, n, n2)$
22: **end if**
23: **end for**
24: **for all** $(v, v2) \in Adj(v)$ **do**
25: **if** $way(v2) = 1$ **and** $bestDist > distR(v) + weight(v, v2) + dist(v2)$ **then**
26: $bestDist \leftarrow distR(v) + weight(v, v2) + dist(v2)$
27: **else**
28: $way(v2) \leftarrow 2$
29: $MiniDijkstraStep(Qp, v, v2)$
30: **end if**
31: **end for**
32: **end while**
32: **end function**=0

```
0: function MiniDijkstraStep(Q, n, n2)
1: if dist(n2) > dist(n) + weight(n, n2) then
2:    dist(n2) ← dist(n) + weight(n, n2)
3:    if n2 ∈ Qs then
4:       Insert(Qs, (n2, dist(n2)))
5:    else
6:       Decrease − Key(Qs, (n2, dist(n2))) {function to decrease the key in Qs}
7:    end if
8: end if
8: end function=0
```

Note: The pseudocode is not complete, missing the parts to calculate the actual path and to calculate the transposed graph (needed for the backward search because this is a directed graph). These parts were excluded in order to not make the algorithm's pseudocode even more complex and spacious.

Analysis

This algorithm basically performs two Dijkstra's algorithms: one starting from the initial node and going forward(lines 16-23); another one starting from the ending node and going backward (lines 24-31). Each time they cross, the distance is registered (lines 25-26 for example). The stopping criteria is a tricky problem in this case. In many situations, the algorithm ends when the two sets collide (forward search and backward search). We found that this criteria did not guarantee the shortest path. Instead, we chose to end the algorithm when there can be no longer any better distances because the nodes being processed are already too far apart.

Efficiency

- **Temporal Complexity** - $O((|N|+|E|) \times \log(|N|)$, where E is the set of edges and N the set of nodes of the graph.
- **Spatial Complexity** - Depends on the implementation, but usually $O(|N|)$

Even though there's a speed up, the complexities will be the same as in Dijkstra's, since the difference is multiplication and division by constants, which is ignored in Big O order.

Usability

Bidirectional Dijkstra's algorithm is one of the fastest algorithms when it comes shortest paths. It has the down-side of requiring the previous knowledge of the ending/destination node, but in this scenario that is not a problem.

4.1.4 A* Algorithms

Description and Analysis

The first time a A* Algorithm came up was in *Shakey Project*, project of the first general purpose mobile robot capable of reason about is own actions. In an A* (pronounced A-star) Search Algorithm, the 'level' of a node in a graph becomes a sum of two different characteristics. For instance, applied to this problem, the distance a node is from the origin in the search algorithm is substituted for the sum of:

- d_{sv} - actual distance to the origin
- π_{nt} - estimated guess of the distance to the goal node (heuristic)

The reliability of this technique depends on the efficacy of the heuristic. In this case, because the weight of edges is distance in km, the efficacy is great if we assume π to be the euclidean distance. This algorithm can be implemented into both Bidirectional and normal Dijkstra's.

- **Upside** - Speeds up the process immensely
- **Downside** - The algorithm does not absolutely guarantee the best solution.

Usability

This is a technique that shows to be very promising and a good idea to implement in the solution of this problem.

4.2 Shortest Path Between All Nodes

This section is dedicated to algorithms that calculate the shortest path between all pairs of points. Such algorithms may be used in the solution, specially in the pre-processing part.

4.2.1 Floyd-Warshall

Description

This algorithm was published by Robert Floyd in 1962, yet it was virtually the same as some algorithms published by Stephen Warshall and Bernard Roy, 1962 and 1959 respectively. It takes an adjacency matrix that represents the graph as an input (empty in the beginning) and calculates the shortest paths between all nodes. The value of a path between to vertices is the sum of all edge's weights through that path. It is registered in the matrix as such: distance of shortest path between 1 and 4 would be matrix[1][4]. Edges may have negative values but the graph can't have a negative cycle.

Extra Data, Data Structures and Algorithms Required

- $|N| \times |N|$ matrix (another one if paths are needed), where $|N|$ is the number of vertices

Algorithm 4 Pseudo-Code for Floyd-Warshall

```
 0: function FloydWarshall(G(N, E))
 1: for all d ∈ M do
 2:    d ← ∞ {M is Matrix of distances}
 3: end for
 4: for all (n1, n2) ∈ E do
 5:    M[n1][n2] ← weight(n1, n2) {E is set of edges}
 6: end for
 7: for i = 0 to |V| do
 8:    for j = 0 to |V| do
 9:       for k = 0 to |V| do
10:          if M[j][k] > M[j][i] + M[i][k] then
11:             M[j][k] ← M[j][i] + M[i][k]
12:          end if
13:       end for
14:    end for
15: end for
15: end function=0
```

Note: The pseudocode is not complete, missing the matrix to calculate the paths and respective instructions. This was done in order to simplify the algorithm and minimize it.

Analysis

First part is basically the preparation of the matrix: fill it with ∞ distances; for the edges existant (lines 1-3), change the values in the matrix to their weight (lines 4-6). Then, all pairs of nodes will be checked on if there is any node the path can pass through that would minimize its cost (lines 7-14).

Efficiency

- **Temporal Complexity** - $O(|N|^3)$, where N is the set of nodes of the graph.
- **Spatial Complexity** - Depends on the implementation, but usually $O(|N|^2)$, for the matrix of size $|N| \times |N|$

Usability

This algorithm is an interesting algorithm to be used in the beginning of the program, to leave all shortest paths processed. However, this might take a long time, and it might be a better idea to simply calculate the shortest path each time it is needed.

4.2.2 Dijkstra's Algorithm

Description and Analysis

Dijkstra's can also be used to calculate all shortest paths. This is done by simply executing the algorithm for every node in the graph as origin.

Efficiency

- **Temporal Complexity** - $O(|N| \times (|N| + |E|) \times \log(|N|)$, where E is the set of edges and N the set of nodes of the graph. Same as Dijkstra's but multiplied by $|N|$
- **Spatial Complexity** - Depends on the implementation, but usually $O(|N|)$, for the priority queue, same as Dijkstra's

Usability

This is a better option for less dense graphs compared to Floyd-Warshall. As this is not the case for a road network, this algorithm is relatively irrelevant.

4.3 Connectivity

The algorithms in this section check the connectivity of the graph. By other words, this section is dedicated to algorithms that detect if any nodes are inaccessible to others, dividing the graph in strongly connected components. A graph will be strongly connected if there is a path between all pairs of points, or if the graph is a strongly connected component itself. Algorithms to test connectivity may be useful to solve the problem of unpredictability.

4.3.1 Brute Force with Floyd-Warshall

Description and Analysis

One way to test the connectivity of a graph is by analyzing the results of Floyd-Warhsall in a certain graph. If there are any pairs of nodes which the distance remains ∞, they do not belong to the same strongly connected component.

Efficiency

- **Temporal Complexity** - $O(|N|^3)$, where N is the set of nodes of the graph, same as Floyd-Warshall
- **Spatial Complexity** - Depends on the implementation, but usually $O(|N|^2)$, for the matrix of size $|N| \times |N|$

Usability

This algorithm works yet is not a good option. Even if the Floyd-Warshall is going to be utilized anyway, the complexity of traversing the whole matrix is $O(|N|^2)$. Therefore, there are much better algorithms for the job.

4.3.2 Kosaraju's algorithm

Description

The algorithm was first published in 1981 by Micha Sharir but gets is name from whom first mencioned it in 1971, Sambasiva Rao Kosaraju. This algorithm allows to identify strongly connected components in a directed graph, testing its connectivity.

Extra Data, Data Structures and Algorithms Required

- boolean identifying if a node has been visited or not (visited)
- identifier of the strong component the node belongs to (strongComponent)
- function to invert the edges of the graph
- a queue to hold the order of the visits

Algorithm 5 Pseudo-Code for Kosaraju's Algorithm

```
 0: function KOSARAJU(G(N, E))
 1: for all n ∈ N do
 2:    visited(n) ← false
 3: end for
 4: strongComponent ← 0
 5: q ← ∅
 6: for all n ∈ N do
 7:    dfsVisit(n, strongComponent, q)
 8: end for
 9: revertEdges(G) {revert all the edges of the graph}
10: for all n ∈ N do
11:    visited(n) ← false
12: end for
13: strongComponent ← 1
14: for all s ≠ ∅ do
15:    dfsVisit(extractFront(s), strongComponent, q)
16:    strongComponent ← strongComponent + 1
17: end for
17: end function
17: function DFSVISIT(n, strongComponent, q) {strongComponent is the identi-
    fier}
18: if visited(n) then
19:    return
20: end if
21: component(n) ← strongComponent
22: if strongComponent = 0 then
23:    Insert(q, n)
24: end if
25: visited(n) ← true
26: for all (n, n2) ∈ Adj(n) do
27:    dfsVisit(n2, strongComponent, q)
28: end for
```

Analysis

The algorithm performs two depth-first searches in the graph. In the first one, it marks the nodes by visit order (lines 5-9). The second one starts on the nodes with

lowest order and is performed on the graph with the edges inverted (lines 10-14). This second one will generate multiple expansion trees (or just one if the graph is strongly connected), each being a strongly connected component. If two nodes are not in the same strongly connected component, one of them cannot be reached from the other.

Efficiency

- **Temporal Complexity** - $O(|N| + |E|)$, where N is the set of nodes of the graph and E the set of edges. Each dfs is $O(|N| + |E|)$
- **Spatial Complexity** - Depends on the implementation, but in this one it is $O(|N|)$, being the size of the queue $|N|$

Usability

This algorithm is a serious candidate to check the connectivity of the graph.

4.3.3 Tarjan's Algorithm

Description

The Tarjan's Algorithm was named after its inventor, Robert Tarjan, and is another algorithm capable of identifying the strongly connected components in a given graph. It does so, similarly to Kosaraju's, with the help of dfs, only this time the graph is only traversed once.

Extra Data, Data Structures and Algorithms Required

- boolean identifying if a node has been visited or not (visited)
- integer identifying the counter when a node is first found (num)
- integer identifying the lowest num reachable by a node (low)

Algorithm 6 Pseudo-Code for Tarjan's Algorithm

```
 0: function TARJAN(G(N, E))
 1: for all n ∈ N do
 2:    visited(n) ← false
 3: end for
 4: numCounter ← 0
 5: for all n ∈ N do
 6:    dfsVisit(n, numCounter)
 7: end for
 7: end function
 7: function DFSVISIT(n, numCounter)
 8: if visited(n) then
 9:    return low(n)
10: end if
11: numCounter ← numCounter + 1
12: num(n) ← numCounter
13: low(n) ← numCounter
14: visited(n) ← true
15: for all (n, n2) ∈ Adj(n) do
16:    temp ← dfsVisit(n2, numCounter)
17:    if temp < low(n) then
18:       low(n) ← temp
19:    end if
20: end for
21: return low(n)
```

Analysis

The algorithm is fairly simple: the graph is traversed by a depth first search (lines 5-6); the *num* and *low* of a node are set to the *numCounter*(lines 12-13); each time an already visited node is found, all nodes with $low > thatnodeslow$ will be changed (lines 16-18). This way, all vertices with the same *low* belong to the same strongly connected component.

Efficiency

- **Temporal Complexity** - $O(|N|+|E|)$, where N is the set of nodes of the graph and E the set of edges. The dfs is $O(|N| + |E|)$. This is the same complexity

as Kosaraju's, but the algorithm is twice as fast since it only performs one dfs.

- **Spatial Complexity** - Depends on the implementation, but in this one it is $O(1)$.

Usability

This algorithm is an even better candidate than the last one, since it is supposedly twice as fast.

4.4 Clustering Algorithms

A Clustering is a machine learning task which involves the recognition of natural grouping in data. Such algorithms can be used in the distribution of orders per vehicle.

4.4.1 Agglomerative Hierarchical Clustering

Description

This algorithm is one of the most common Hierarchical Clustering algorithms. It is used to form groups of objects of data based on their similarity. In this case, we will use it to form N groups of data nodes with great similarity.

Algorithm 7 Pseudo-Code for Clustering algorithm

```
0: function AHC(S, n) {S is a set points, n is the number of final clusters}
1: SS ← GenerateSuperSet(S) {Generate a set of sets, initially each subset has
   one of S's points}
2: while size(SS) > n do
3:    bestDist ← ∞
4:    bestDistPointA ← nil
5:    bestDistPointB ← nil
6:    for all p ∈ SS do
          {v and p are sets} for all v ∈ SS do
8:        if v ≠ p and dist(v, p) < bestDist then
9:           bestDist ← dist(v, p)
10:          bestDistPointA ← v
11:          bestDistPointB ← p
12:       end if
13:    end for
14:   end for
15:   join(bestDistPoinA, bestDistPointB)
16: end while
16: end function=0
```

Note: the implementation presented above is just a rough sketch, the one to be implemented will be different.

Analysis

The algorithm begins to find the pair of points which are most similar. In this case, similarity is measured by distance (lines 6-14). After he has found one, he merges the two sets (. In between sets, there are many criteria to calculate similarity. The one we will implement is taking the similarity between two sets as the similarity of the most similar pair of points where one belong to the first set and other to the second set. This algorithm adopts a Bottom-Up strategy (as do all agglomerative hierarchical clustering algorithms), as it start by processing each node and joining them to obtain the final result.

Efficiency

- **Temporal Complexity** - $O(|S|^2 * (|S| - n))$, where S is the set of points. This is only an estimated guess.
- **Spatial Complexity** - Depends on the implementation, but in this one it is $O(|S|)$.

Usability

This is the only algorithm analysed in this cathegory, thus it is the chosen one to perform the task.

4.5 Extras

There are other algorithms, simpler or less relevant, that may (or may not) be used in our solution:

- Breadth-First Search (bfs) - Graph searching algorithm
- Depth-First Search (dfs) - Graph searching algorithm
- Decrease-Key Function - Priority changing function on a priority queue
- Graph Transposing Function - Inversion of the edges in a directed graph
- Fibonacci Heaps (Data structure) - Improved priority queue
- Euclidian Distance function - To calculate the euclidian (straight-line) distance between two points

Chapter 5

Prospective Solution

5.1 Pre-Processing of Input Data

Before going through with the problem resolution and trying to find the shortest path for the vehicles, we opt to pre-process the graph, making the other tasks easier.

- calculating the shortest paths for all pairs of nodes in a certain radius of the company's garage through use of Floyd-Warshall's algorithm. This will allow the shortest paths between most used nodes to already be calculated, speeding up the calculation of itineraries
- evaluating the connectivity of the graph through the Tarjan's algorithm, followed by removal of the irrelevant elements

5.2 Problem Identification

With the graph and the orders pre-processed, there's now a need to find the most efficient way for the vehicles to pick up and deliver the costumers' orders. Before, we have divided the problem into four problems:

- **I** - Calculation of the itineraries
- **II** - Inclusion of recharging points in the paths as needed
- **III** - Evaluation of the graph's connectivity
- **IV** -Distribution of the orders between the vehicles available to the company

5.3 Solutions

5.3.1 Problem I - Itineraries/Routes Calculation

Although most relevant pairs of nodes already have their shortest path calculated, some pick up or delivery points might be out of bound for the pre-processing calculations. For these and other special cases, we deemed Bidirectional Dijkstra with implementation of A* algorithm the best option. This choice was made considering the speed of the algorithm and the fact that the destination node is supposedly always known.
The itinerary for a vehicle is calculated having into account the shortest paths between the interest points of his orders (and the recharge points). The algorithm to perform this task will most likely implement a backtracking strategy, in case autonomy is a problem.

5.3.2 Problem II - Autonomy/Range

To tackle this problem, we use the following strategy:

1. Before every assignment of a vehicle to a certain path, its current range must be evaluated.
2. If the value is not enough for the voyage, the vehicle must localize one or more recharge points. The criteria to find the best points is to choose the recharge point closest to both the starting and the finishing node
3. If the value is enough, it must also be checked if the vehicle has enough autonomy to get to a recharge point after the path
4. If both 2. and 3.'s answers are positive, it is safe to proceed. Else, recharge point must be included in his itinerary
5. If there are no recharge points available in the conditions, the itinerary is aborted

5.3.3 Problem III - Connectivity

Tarjan's algorithm was our choice to solve the connectivity problem. By checking the connectivity of the graph, we can check for any nodes that are not accessible, supposedly due to road work, accidents or other unexpected events. We chose Tarjan's because it presents itself as the fastest option from the ones we analysed.

5.3.4 Problem IV - Orders Distribution

The distribution of orders by the vehicles is by far the most challenging subproblem. This is a Rout Optimization problem. Our solution for it was not optimal, yet we made an effort to make the result the best possible. We solved this problem by dividing the pick up points (representing each a different order) into $|V|$ different groups, where $|V|$ is the number of vehicles available. We do this using the Agglomerative Hierarchical Clustering described in the previous section.

5.3.5 General

In sum, our solution:

1. Pre-Processing of the graph in terms of shortest paths and connectivity
2. Ditribution of Orders into groups
3. Calculation of the best rout for each group (which represents a vehicle's orders)
    - Use of the shortest paths pre-calculated or calculate new ones if necessary
    - Try the best rout; if autonomy deems into impossible at any point, rewind until you can change a decision (backtracking)

Note: The algorithms and solutions presented may not be the exact same to be used, they only represent an estimation. In the second part of this work, the algorithms will be tested and some changes may be necessary.

Chapter 6

Functionalities Implemented

To aid in the usage of the program, an interface will be developed. This interface will enable the user to perform many actions, both related to the graph's processing and to information about orders, clients, etc.

Graph Related

- Redo the check to the graph's connectivity (and consequently update the graph if necessary)
- Calculate the paths for each vehicle in that day, considering the existing orders
- Add or remove a pick up point
- Add or remove a delivery point

Orders and Clients

- Add or remove an order (a Client will automatically be removed if he no longer has orders registered to him, past or present)
- Add or remove a client from the clients' list, given he does not have any current order (consequently removing the past orders)
- List all orders, their Clients and the pick up and delivery points
- List past orders, their Clients and the pick up and delivery points
- List/Search orders by other filters

- List all clients
- List/Search clients by certain filters

Vehicles and Drivers

- List all vehicles and some brief information about them
- Add or remove vehicles
- Specifically search a vehicles (most likely by its license plate)
- List all drivers and some brief information about them
- Add (hire) or remove (fire) a driver
- Specifically search for a driver

This list only presents a preview of the program's interface, more or less functionalities may be implemented.

Chapter 7

Conclusion

In sum, the task at end was considered to be fulfilled with success. The problem was formalized and divided into subproblems, which were accessed individually. In each assessment, a research for a possible solution was carried out, originating the possible algorithms to use. Each algorithm was carefully analysed, in order to decide whether it was a suitable candidate for each job. Finally a solution was fabricated, compiling multiple algorithms and a final strategy for the problem.

7.1 Tasks allocation

Participants	**Marcelo**	**João**	**Afonso**
Problem Description	All	-	-
Problem Formalization	All	-	-
Algorithms	Most	Small contribution	-
Prospective Solution	Most	Small contribution	-
Functionalities	Most	Small contribution	-
Conclusion	All	-	-
Investigation	Most	Decent Contribution	-

Chapter 8

Bibliography

- **Clustering Algorithm**
  - https://en.wikipedia.org/wiki/Hierarchical_clustering
  - https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68
- **Bidirectional Dijkstra's**
  - https://www.geeksforgeeks.org/bidirectional-search/
  - https://www.youtube.com/watch?v=KyRSJRMz818&t=1401s
  - slides by R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão CAL, MIEIC-FEUP
  - https://www.homepages.ucl.ac.uk/~ucahmto/math/2020/05/30/bidirectional-dijkstra.html
- **A* algorithm**
  - https://www.geeksforgeeks.org/a-search-algorithm/
  - slides by R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão CAL, MIEIC-FEUP
- **Connectivity algorithms**
  - https://www.geeksforgeeks.org/strongly-connected-components/
  - https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/

  - slides by R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão CAL, MIEIC-FEUP
  - `https://www.youtube.com/watch?v=Rs6DXyWpWrI`
  - `https://iq.opengenus.org/tarjans-algorithm/`

- **Other algorithms and information**
  - slides by R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão CAL, MIEIC-FEUP