ROUTE OPTIMIZATION ALGORITHM FOR RIDESHARING

Maria Sofía Uribe
Universidad EAFIT
Colombia
msuribec@eafit.edu.co

Isabel Cristina Graciano
Universidad EAFIT
Colombia
icgracianv@eafit.edu.co

Mauricio Toro
Universidad Eafit
Colombia
mtorobe@eafit.edu.co

ABSTRACT
The present document gives a literature review of problems related to optimization algorithms for ridesharing routes. The broader problem is the negative effects brought by the surplus of cars present in most cities today. Solving this problem would have a positive impact in the environment and traffic congestion

KEYWORDS: ROUTE OPTIMIZATION, VEHICLE ROUTING PROBLEM, TSP PROBLEM, GRAPH CLUSTERING, ALGORITHMIC ANALYSIS.

ACM CLASSIFICATION Keywords

Shortest paths → Graph algorithm analysis → Graph algorithms analysis → Paths and connectivity problems → Graph algorithms.

1. INTRODUCTION
Traffic congestion is one the most substantial challenges for crowded cities, in environmental terms reducing greenhouse gases like Co2 is a necessity to improve quality of air in urban areas. Having more active vehicles also means traffic jams become frequent, while public transportation may subdue the overall number of automobiles in many cases it is not enough to solve the problem completely.

Solutions such as introducing driving restrictions or congestion fees have been proposed in multiple cities around the globe. According to the 2018 inrix global traffic scorecard, drivers in some cities can spends upwards of 100 hours per year sitting in traffic, which is a waste of time and money, as an example in London the estimated cost of congestion per driver is 1,680 British pounds.

Carpooling, sharing a car ride within a group of users, can offer benefits like decreasing traffic jams, fuel cost and carbon dioxide emission per person.

2. PROBLEM

Proposing carpooling to reduce the number of cars on the road is not enough. In order to make carpooling an attractive solution, one must provide all people with greater benefits than the alternative of going alone in their own car, that is if a person chooses to pick people that live near them they should not have to drastically modify the current route they take to work or time that it takes them to reach their destination.

3. RELATED WORK

3.1 Vehicle routing problem

The VRP definition states that a fleet of m vehicles with a start point a must deliver goods to n customers, given that the quantity of goods is discrete. The objective is to minimize the overall transportation cost. provided that each there will be only one delivery per location. The best route distribution is the one that reduces the number of required vehicles and the time of each route.

Given that in real life applications other constraints like capacity, time windows and precedence relations exist . the term VRP is used to describe a family of similar problems and not just one specific scenario.

A constrained version is the CVRP. In a CVRP, each location has a demand—a physical quantity, such as weight or volume, corresponding to an item to be picked up or delivered there. Each time a vehicle visits a location, the total amount the vehicle is carrying increases (for a pickup) or decreases (for a delivery) by the demand at that location. In addition, each vehicle has a maximum capacity for the total amount it can carry at any time.

A CVRP can be represented by a graph with distances assigned to the edges and demands assigned to the nodes.

Solutions

There are exact algorithms that won't find optimal routes for many cities, most solutions are heuristic algorithms. Some smarter exact approaches like branch and bound uses a divide and conquer strategy to partition the solution space into subproblems and then optimizes individually over each subproblem. If it is found that the subset has no solution or that the best possible solution within said subset won't be better than the global upper bound, the entire subset or branch is discarded. In case the subproblem cannot be discarded the algorithm, branches and adds the children of this subproblem to the list of active candidates. This cycle continues until the list of active candidates is empty and if a solution that's better than the current one is found; the optimal solution is updated.

Graphics

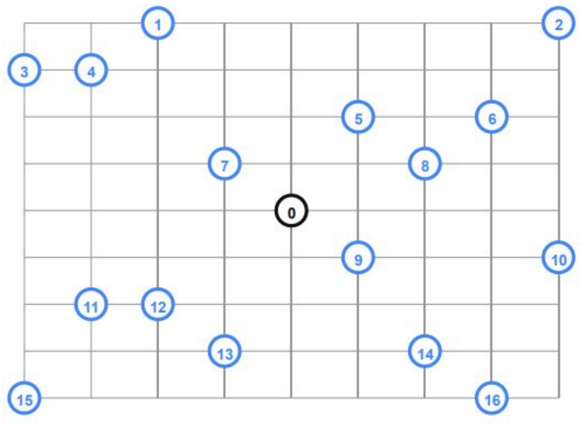


Figure 1 : A diagram of locations in blue that need to be visited from the source 0. Retrieved from [6]

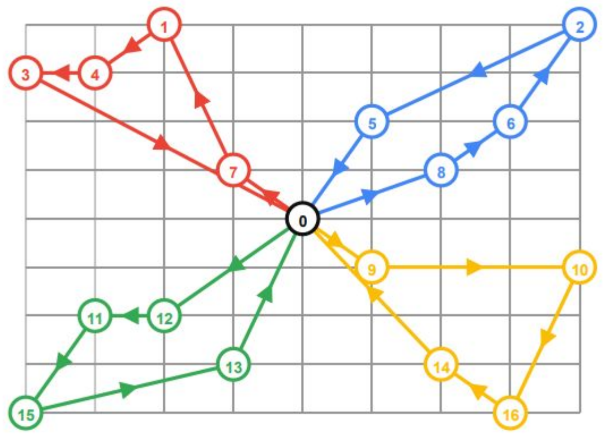


Figure 2 :Assigned routes to the former graph. Retrieved from [6]

3.2 Network connections. Minimum spanning tree.

A common graphs problem is the design of network connections. Given a set of k places it is necessary to link them to a source using the shortest length of cable due to its price. All locations must be connected but it is not necessary that they be connected directly. If it is constrained to bury the cable only along certain paths, then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. The concept used to solve this problem is a minimum spanning tree which we discuss below, these trees can help with problems that use techniques such as cluttering and matching.

Solution

A solution to this problem can be achieved with Kruskal's algorithm, a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Each iteration it finds an edge which has least weight and adds it to the growing spanning tree. First it sorts the graph edges with respect to their weights and then it starts adding edges to the MST from the edge with the smallest weight until the edge of the largest weight (only adding edges that do not form cycles).

Graphics

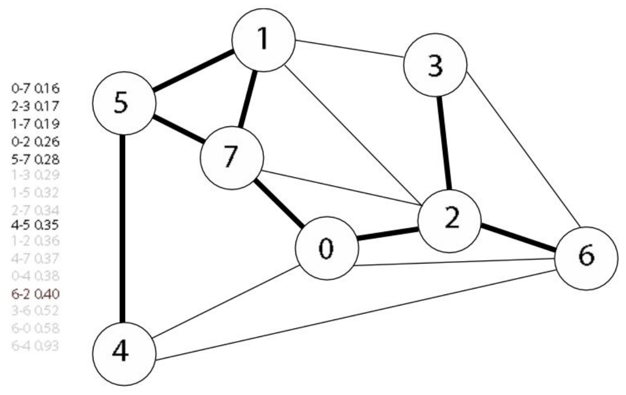


Figure 3 : MST generated with Kruskal's algorithm. Adapted from [10].

3.3 The travelling salesman problem.

The traveling salesman problem consists of a salesman and a set of cities. The salesman must visit each one of the cities starting from a certain one (e.g. the hometown) and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip.

The traveling salesman problem can be described as follows:

$$TSP \ = \ \{(G, f, \ t):$$

$$G = (V\,, E) \text{ a complete graph,}$$

$$f \text{ is a function } V \times V \ \rightarrow Z, \ t \in Z,$$

$$G \text{ is a graph that contains a traveling salesman tour with cost}$$

$$\text{that does not exceed } t\}.$$

Solution

There is generally no known best method of solving this problem, it is NP-hard (Nondeterministic Polynomial-time hard) problem. A heuristic solution proposed by Karp is to partition the problem to get an approximate solution using the divide and conquer techniques. We form groups of the cities and find optimal tours within these groups. Then we combine the groups to find the optimal tour of the original problem. The deviation from the optimal solution will depend largely on the mechanism used to divide the problem, the most straightforward approach is to group cities in terms of their location. A less obvious approach is to divide the cities in equal sized cells

Graphics

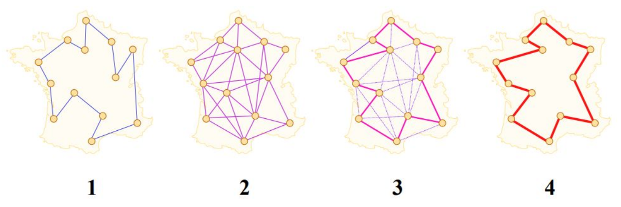


Figure 4: The ant colony optimization of the travelling salesman problem. Retrieved from [11]

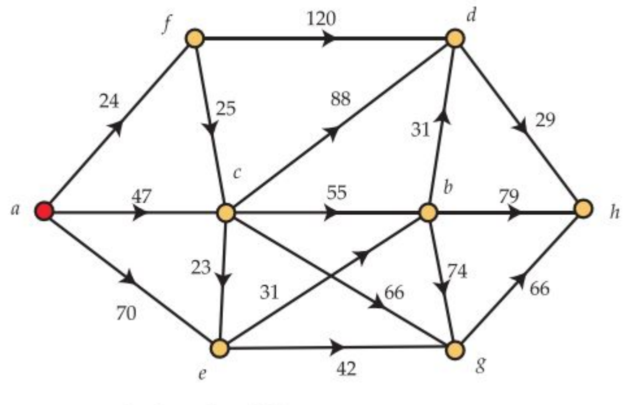
3.4 Shortest path problem

The problem of finding the shortest path (a.k.a. graph geodesic) connecting two specific vertices (u, v) of a directed or undirected graph. The length of the graph geodesic between these points d (u, v) is called the graph distance between u and v.

Solution

Common algorithms for solving the shortest path problem include the Bellman-Ford algorithm and Dijkstra's algorithm. The latter one functions by constructing a shortest-path tree from the initial vertex to every other vertex in the graph. The algorithm maintains a priority queue minQ that is used to store the unprocessed vertices with their shortest-path estimates est(v) as key values. It then repeatedly extracts the vertex u which has the minimum est(u) from minQ and relaxes all edges incident from u to any vertex in minQ. After one vertex is extracted from minQ and all relaxations through it are completed, the algorithm will treat this vertex as processed and will not touch it again. Dijkstra's algorithm stops either when minQ is empty or when every vertex is examined exactly once.

Graphics



$\sigma = (a, f, c, e, b, g, d, h)$

$\delta(a) = 0;$	$P(a) = (a)$
$\delta(b) = 101;$	$P(b) = (a, e, b)$
$\delta(c) = 47;$	$P(c) = (a, c)$
$\delta(d) = 132;$	$P(d) = (a, e, b, d)$
$\delta(e) = 70;$	$P(e) = (a, e)$
$\delta(f) = 24;$	$P(f) = (a, f)$
$\delta(g) = 112;$	$P(g) = (a, e, g)$
$\delta(h) = 161;$	$P(h) = (a, e, b, d, h)$

Figure 5: Results of Dijkstra's Algorithm. Retrieved from [12].

4. ROUTE OPTIMIZATION ALGORITHM USING A* ALGORITHM

We used a Hash function in which the key will has ID node and the value has the node; In addition, each node is composed for a set of LinkedList of edges so every edge has to have the information about its weight and destination.

4.1 Data Structure design

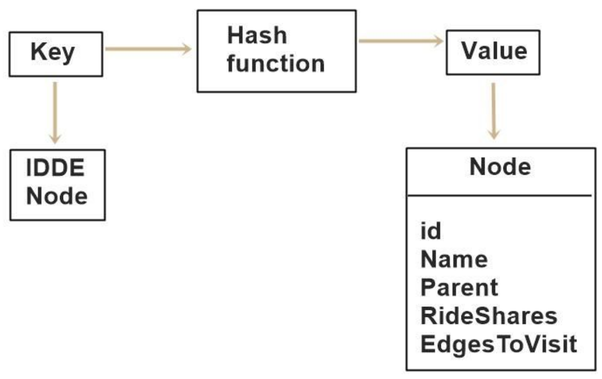


Figure 1: Data structure of the HashMap.

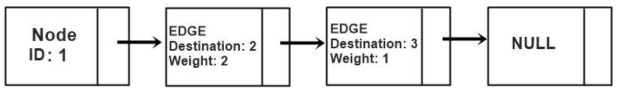


Figure 1: Data structure of the LinkedList.

4.2 Design of the operations of the data structure

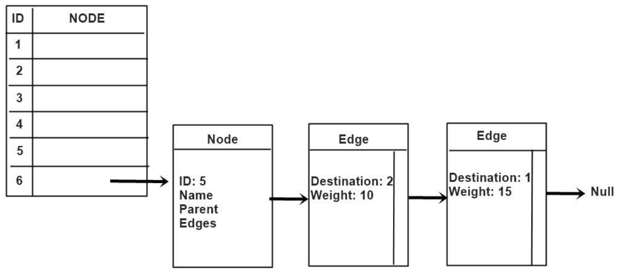


Figure 2: Gets the cost from a node to another one.

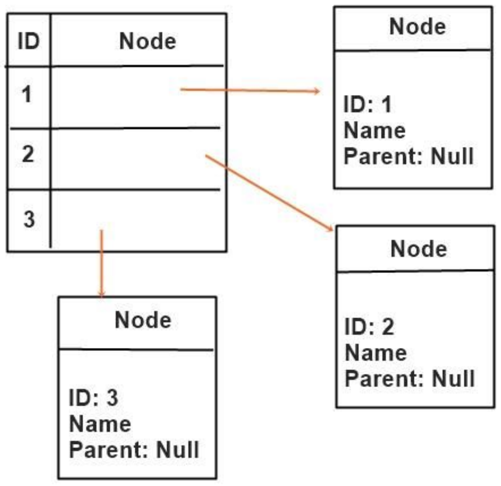


Figure 3: Set the parent of every node to null in the hash table.

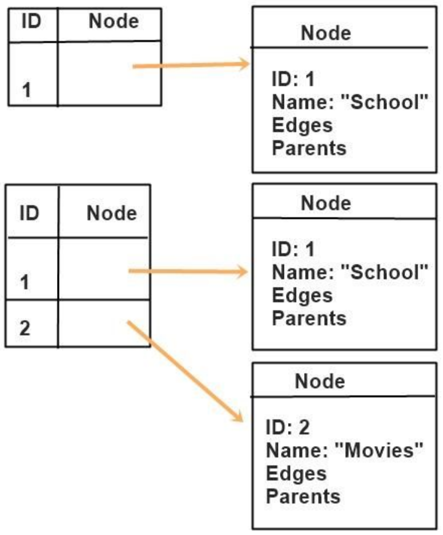


Figure 4: Adds a node.

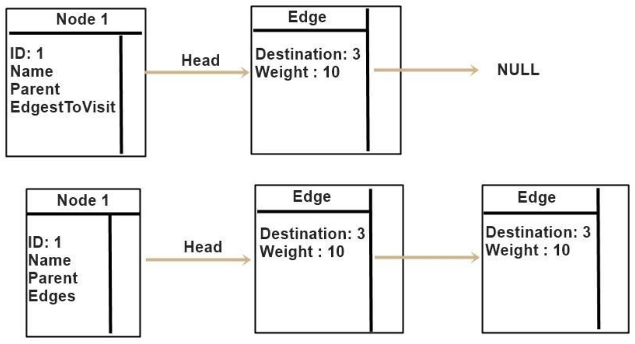


Figure 5: Add successor.

4.3 Design criteria of the data structure

The chosen data structure to represent the graph is a Hash table where the mapping key is the id of the node and the value is the reference to the node with that id.

Given the size of data that must be processed memory usage was a concern, using hash tables allows us to reduce the memory usage and also the time complexity of operations like retrieving a value and putting a value into the structure, given that we have no collisions the hash table can perform these operations in O(1).

Each node has a list of reachable edges, each edge contains the destination and the weight. In this case a Linked List was a good option because it provides iterators which can help the performance when traversing the reachable edges of a node.

4.4 Complexity analysis

Method	Complexity

AddNode	O(1)
Add Successor	O(1)
GetMinCost	O(V)
reset	O(V)

Table 1: Table to report complexity analysis

Where V is the number of nodes in the graph

4.5 Execution time

Number of cars and p	5 cars P=1.2	5 cars P=1.7	205 cars P=1.1	205 cars P=1.2	205 cars P=1.3
Best case	0 ms	0 ms	100	95	120 ms
Average case	1 ms	1 ms	142	139 ms	190 ms
Worst Case	5 ms	6 ms	188 ms	190 ms	234 ms

Table 2: Execution time of the operations of the data structure for each data set.

4.6 Memory consumption

Dataset	5 cars P=1.2	5 cars P=1.7	205 cars P=1.1	205 cars P=1.2	205 cars P=1.3
Memory Consumption	0 MB	0 MB	8 MB	10 MB	11 MB

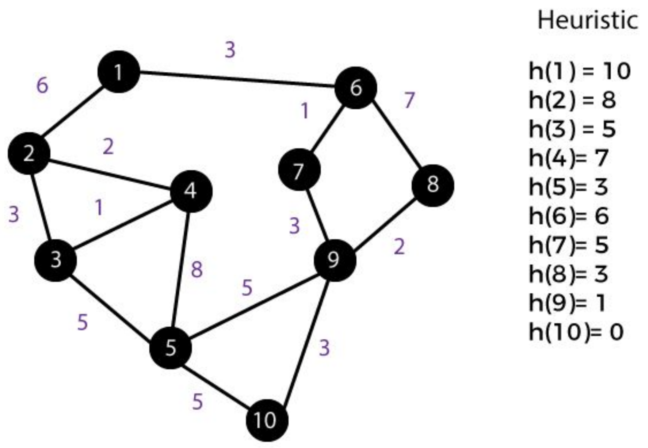
Table 3: Memory used for each operation of the data structure and for each data set data sets.

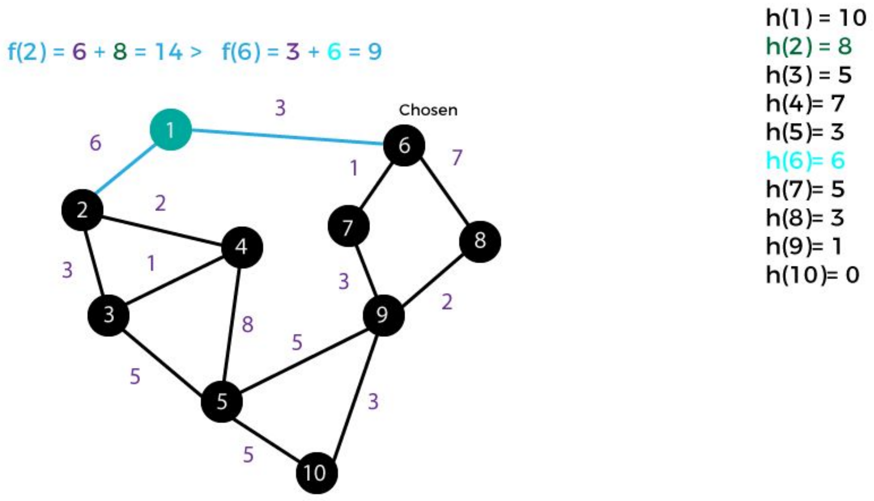
4.7 Result analysis

	Linked List	HashMap
HeapSpace	90 MB	165 MB
Time of creation	147 ms	50 ms
Time of lookup	15 ms	7 ms

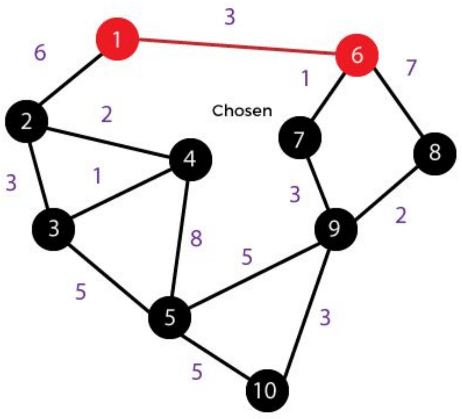
Table 4: Analysis of the results

4.8 Algorithm



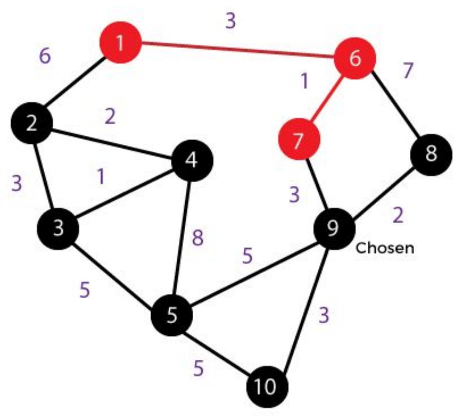


$f(7) = 4 + 5 = 9 < f(8) = 10 + 3 = 17$



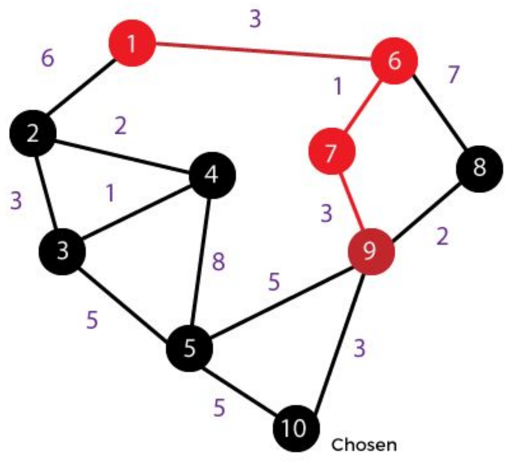
h(1) = 10
h(2) = 8
h(3) = 5
h(4)= 7
h(5)= 3
h(6)= 6
h(7)= 5
h(8)= 3
h(9)= 1
h(10)= 0

$f(9) = 7 + 1 = 8$

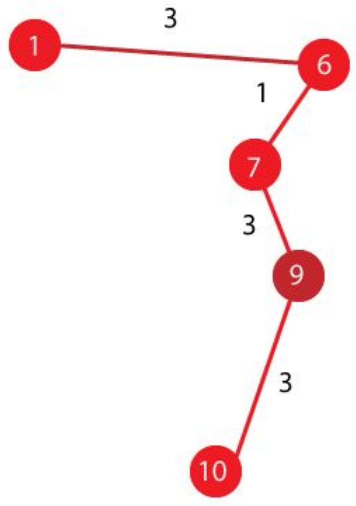


$h(1) = 10$
$h(2) = 8$
$h(3) = 5$
$h(4) = 7$
$h(5) = 3$
$h(6) = 6$
$h(7) = 5$
$h(8) = 3$
$h(9) = 1$
$h(10) = 0$

$f(10) = 10 + 0 = 10 <$ $f(8) = 9 + 3 = 12 <$ $f(5) = 12 + 3 = 15$



$h(1) = 10$
$h(2) = 8$
$h(3) = 5$
$h(4) = 7$
$h(5) = 3$
$h(6) = 6$
$h(7) = 5$
$h(8) = 3$
$h(9) = 1$
$h(10) = 0$



Because the path we found is the shortest path from 1 to 10 ,
10 will ride with 9,7 and 6

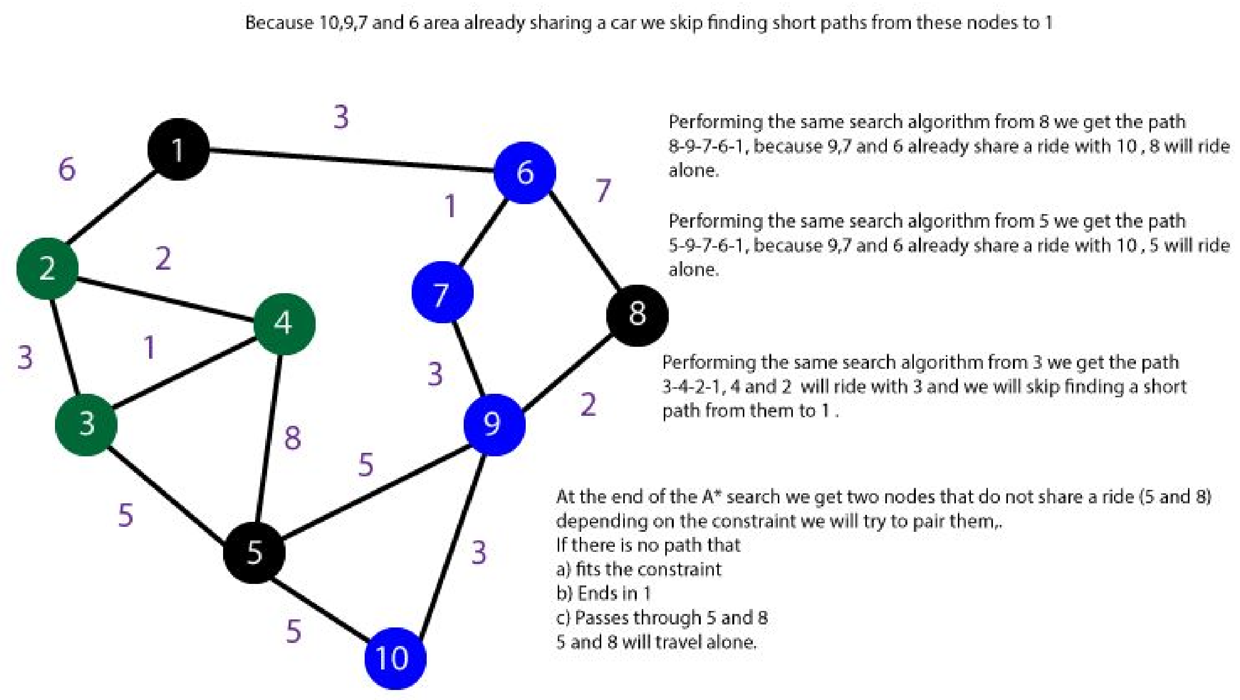

Because 10,9,7 and 6 area already sharing a car we skip finding short paths from these nodes to 1
Performing the same search algorithm from 8 we get the path 8-9-7-6-1, because 9,7 and 6 already share a ride with 10 , 8 will ride alone.
Performing the same search algorithm from 5 we get the path 5-9-7-6-1, because 9,7 and 6 already share a ride with 10 , 5 will ride alone.
Performing the same search algorithm from 3 we get the path 3-4-2-1, 4 and 2 will ride with 3 and we will skip finding a short path from them to 1 .
At the end of the A* search we get two nodes that do not share a ride (5 and 8) depending on the constraint we will try to pair them,.
If there is no path that
a) fits the constraint
b) Ends in 1
c) Passes through 5 and 8
5 and 8 will travel alone.
1
2
3
4
5
6
7
8
9
10
3
6
7
1
2
2
3
1
3
8
5
5
5
3

Figure 3: Step by step explaining how the algorithm assigns rides

4.9 Complexity analysis of the algorithm

Subproblem	Worst Case Complexity
Creating the graph (Reading the file and creating the graph)	O(V+E)
Search(A*) (Reconstruct one short path to 1)	O(V+E)
Assign Vehicles	O(V^2+EV)
Total complexity	O(V^2+EV)

Table 5: Complexity of each subproblem that is part of the algorithm. Where V is the number of nodes of the graph an E is the number of edges

4.10 Design criteria of the algorithm

After evaluating the performance of different algorithms, we found that the A* algorithm provides a good time complexity if the heuristic h that is used is a good lower bound for the path between a node u and the goal node, in this case node 1.

Because we already have a list of short paths computing a good heuristic is relatively easy given the datasets. This would be an improvement on an algorithm purely based on breadth first search or Dijkstra. The current implementation is Closer to a greedy best first search.

The current solution is an approximation, it reconstructs a short path for some of the nodes that are further away and then after this search the algorithm pairs some of the nodes that aren't sharing a car, provided the condition is not violated.

In order to improve the current solution, the algorithm could preprocess the graph to compute the most optimal node from where to start.

4.11 Execution times

	Dataset 1(ms)	**Dataset 2(ms)**	**Dataset 3(ms)**	**Dataset 4(ms)**	**Dataset 5(ms)**
Best case	**0**	**0**	**390**	**300**	**343**
Average case	**15**	**12**	**450**	**395**	**429**
Worst Case	**20**	**21**	**586**	**500**	**570**

Table 6: Execution time of the algorithm for different datasets.

4.12 Memory consumption

Datatset	5 cars P=1.2	5 cars P=1.7	205 cars P=1.1	205 cars P=1.2	205 cars P=1.3
Memory Consumption	0 MB	0 MB	20 MB	22 MB	20 MB

Table 7: Memory consumption of the algorithm for different datasets.

4.13 Analysis of the results

	Linked List	HashMap
HeapSpace	10 MB	25 MB
Time for Directed search	200ms	100ms
Time for Directed search and assigning vehicles	400ms	200ms

Table 8: Analysis of the results obtained from the algorithm execution.

5.

ROUTE OPTIMIZATION ALGORITHM USING A* ALGORITHM AND ADJACENCY MATRIX

We used an adjacency matrix that stores Edges; an edge object represents the arc connecting two nodes and also stores the time it takes to go from the source node to the destination node (in minutes) and the distance in kilometers between the nodes.

5.1 Data Structure design

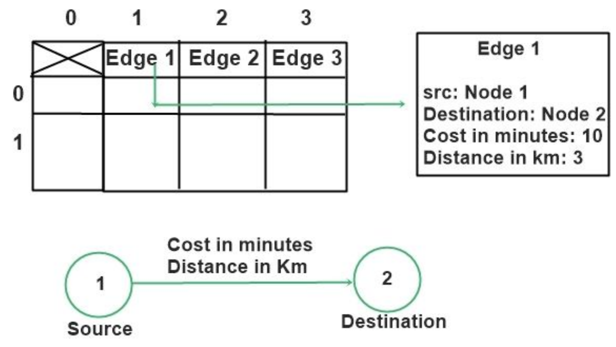


Figure 1: Data structure of the Matrix.

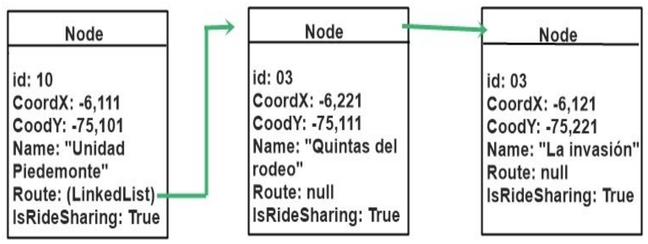


Figure 2: Data structure of the list that contains each node's route. Every car is represented as a node and some cars will pick up other people, therefore, cars (or nodes) will have a route indicating who is going to take to the college but those who are going to be picked up by other car don't need to have a route.

5.2 Design of the operations of the data structure

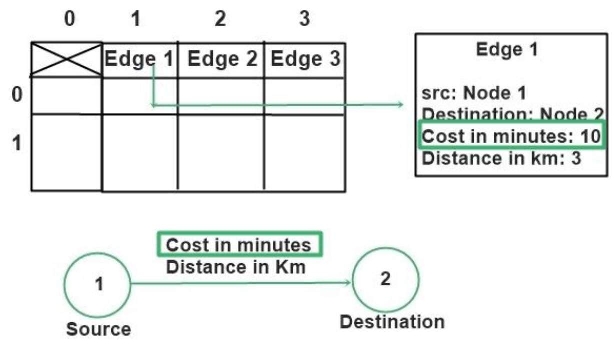


Figure 3: Gets the cost in minutes from a node to another one going to the given location in the matrix and returning the cost.

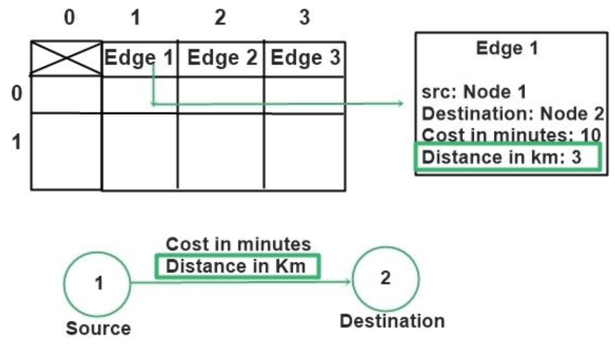


Figure 4: Gets the distance in kilometers from a node to another one going to the given location in the matrix and returning the distance.

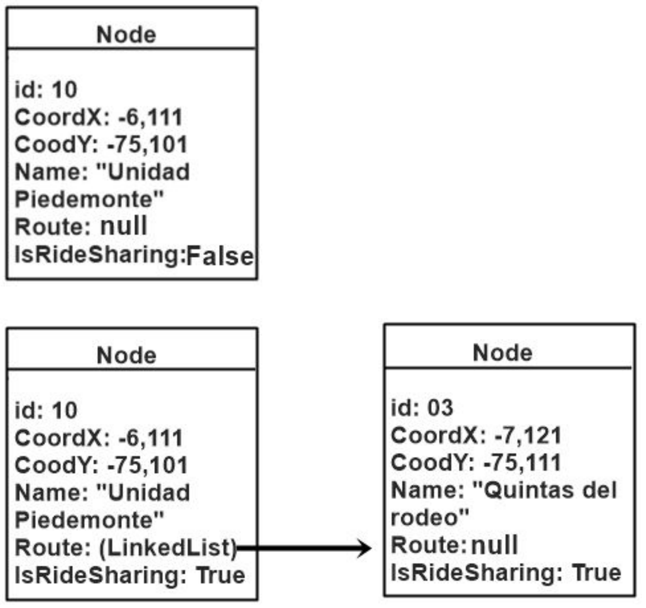


Figure 5: Adds a node to another node's route.

5.3 Design criteria of the data structure

The chosen data structure to represent the graph is an adjacency matrix where we store edges that connect two nodes. Each edge contains the source node, destination node, the cost in minutes and the distance in Kilometers.

Given that the size of the data is not too large, using a matrix is a good enough solution, also it will provide us access to the nodes in O (1) time.

Each node has a list of nodes that have been assigned to its route, In this case a Linked List was a good option because it provides iterators which can help the performance when traversing the reachable children nodes (cars being picked up) of a parent node(main car).

5.4 Complexity analysis

Method	Complexity
addCartoRoute	O(1)
GetCostMinutes	O(1)
GetDistanceKM	O(1)
reset	O(V)

Table 1: Table to report complexity analysis

Where V is the number of nodes in the graph

5.5 Execution time

Number of cars and p	5 cars P=1.2	5 cars P=1.7	205 cars P=1.1	205 cars P=1.2	205 cars P=1.3	205 cars c= 6.1
Best case	0 ms	0 ms	100	95	120 ms	47 ms
Average case	1 ms	1 ms	142	139 ms	190 ms	50 ms
Worst Case	5 ms	6 ms	188 ms	190 ms	234 ms	62 ms

Table 2: Execution time of the operations of the data structure for each data set.

5.6 Memory consumption

Dataset	5 cars P=1.2	5 cars P=1.7	205 cars P=1.1	205 cars P=1.2	205 cars P=1.3	205 cars c= 6.1
Memory Consumption	0 MB	0 MB	8 MB	10 MB	11 MB	12 MB

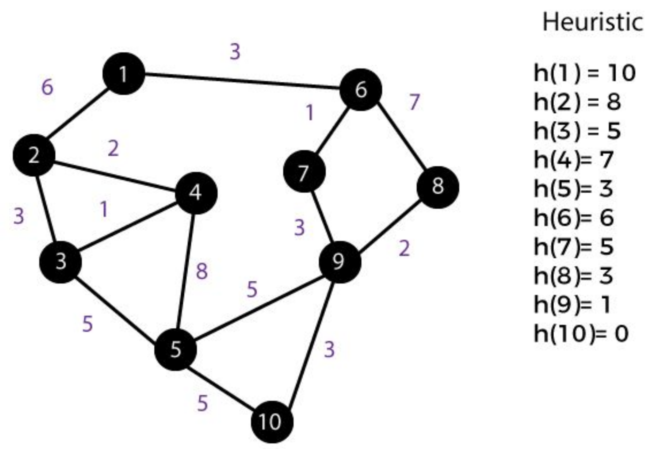
Table 3: Memory used for each operation of the data structure and for each data set data sets.

5.7 Result analysis

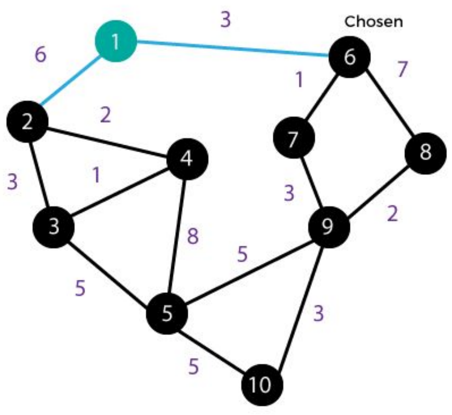
	Linked List	HashMap	Matrix
HeapSpace	90 MB	165 MB	110MB
Time of creation	147 ms	50 ms	47 ms
Time of lookup	15 ms	7 ms	7 ms

Table 4: Analysis of the results

5.8 Algorithm

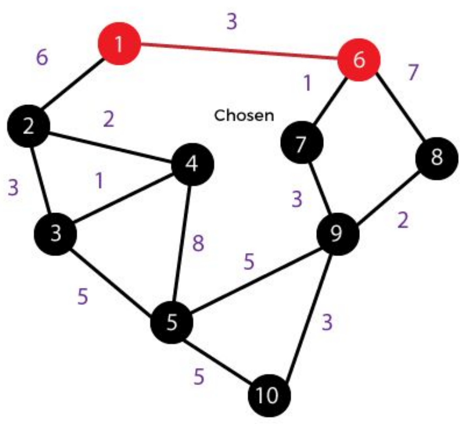


$f(2) = 6 + 8 = 14 > f(6) = 3 + 6 = 9$



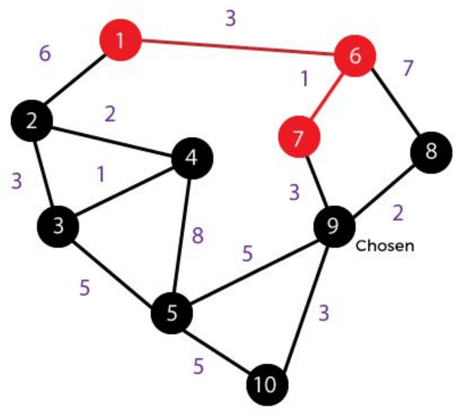
h(1) = 10
h(2) = 8
h(3) = 5
h(4)= 7
h(5)= 3
h(6)= 6
h(7)= 5
h(8)= 3
h(9)= 1
h(10)= 0

$f(7) = 4 + 5 = 9 < f(8) = 10 + 3 = 17$



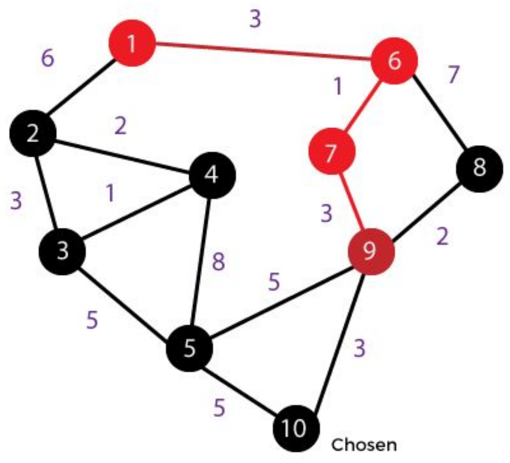
h(1) = 10
h(2) = 8
h(3) = 5
h(4)= 7
h(5)= 3
h(6)= 6
h(7)= 5
h(8)= 3
h(9)= 1
h(10)= 0

$f(9) = 7 + 1 = 8$

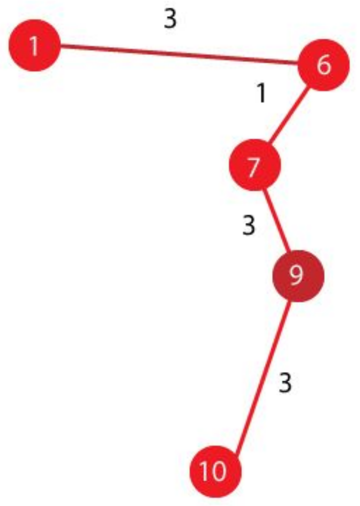


h(1) = 10
h(2) = 8
h(3) = 5
h(4)= 7
h(5)= 3
h(6)= 6
h(7)= 5
h(8)= 3
h(9)= 1
h(10)= 0

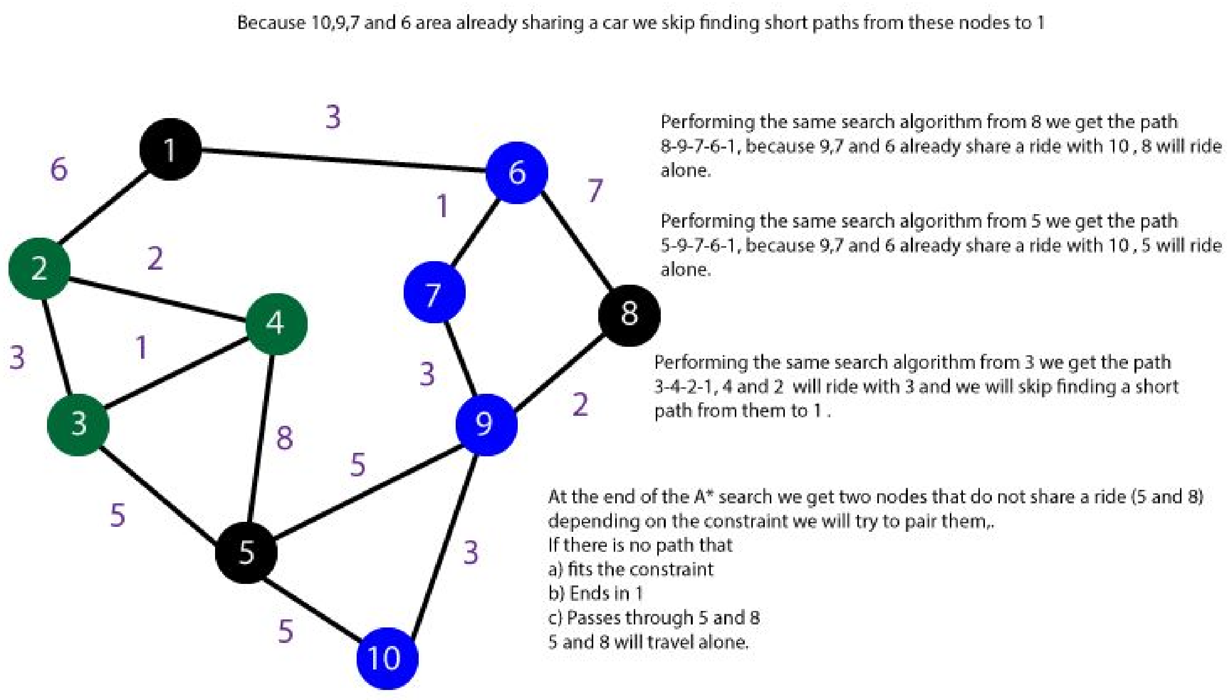
$f(10) = 10 + 0 = 10 <$ $f(8) = 9 + 3 = 12 <$ $f(5) = 12 + 3 = 15$



h(1) = 10
h(2) = 8
h(3) = 5
h(4)= 7
h(5)= 3
h(6)= 6
h(7)= 5
h(8)= 3
h(9)= 1
h(10)= 0



Because the path we found is the shortest path from 1 to 10 ,
10 will ride with 9,7 and 6


Because 10,9,7 and 6 area already sharing a car we skip finding short paths from these nodes to 1

Performing the same search algorithm from 8 we get the path 8-9-7-6-1, because 9,7 and 6 already share a ride with 10 , 8 will ride alone.

Performing the same search algorithm from 5 we get the path 5-9-7-6-1, because 9,7 and 6 already share a ride with 10 , 5 will ride alone.

Performing the same search algorithm from 3 we get the path 3-4-2-1, 4 and 2 will ride with 3 and we will skip finding a short path from them to 1 .

At the end of the A* search we get two nodes that do not share a ride (5 and 8) depending on the constraint we will try to pair them,.
If there is no path that
a) fits the constraint
b) Ends in 1
c) Passes through 5 and 8
5 and 8 will travel alone.

Figure 3: Step by step explaining how the algorithm assigns rides

5.9 Complexity analysis of the algorithm

Subproblem	Worst Case Complexity
Creating the graph (Reading the file and creating the graph)	O(V+E)
Directed Search(A*) (Reconstruct one short path to 1)	O(V+E)
Assign Vehicles	O(V^2+EV)
Total complexity	O(V^2+EV)

Table 5: Complexity of each subproblem that is part of the algorithm. Where V is the number of nodes of the graph an E is the number of edges

5.10 Design criteria of the algorithm

After evaluating the performance of different algorithms, we found that the A* algorithm provides a good time complexity if the heuristic h that is used is a good lower bound for the path between a node u and the goal node, in this case node 1.

Because we already have a list of short paths computing a good heuristic is relatively easy given the datasets. This would be an improvement on an algorithm purely based on breadth first search or Dijkstra. The current implementation is Closer to a greedy best first search.

The current solution is an approximation, it reconstructs a short path for some of the nodes that are further away and then after this search the algorithm pairs some of the nodes that aren't sharing a car, provided the condition is not violated.

In order to improve the current solution, the algorithm could preprocess the graph to compute the most optimal node from where to start.

5.11 Execution times

	5 cars P=1.2	5 cars P=1.7	205 cars P=1.1	205 cars P=1.2	205 cars P=1.3	205 cars C= 6.1
Best case	**0**	**0**	**390**	**300**	**343**	**300**
Average case	**15**	**12**	**450**	**395**	**429**	**349**
Worst Case	**20**	**21**	**586**	**500**	**570**	**370**

Table 6: Execution time of the algorithm for different datasets.

5.12 Memory consumption

Datatset	5 cars P=1.2	5 cars P=1.7	205 cars P=1.1	205 cars P=1.2	205 cars P=1.3	205 cars C= 6.1
Memory Consumption	0 MB	0 MB	20 MB	22 MB	20 MB	20 MB

Table 7: Memory consumption of the algorithm for different datasets.

5.13 Analysis of the results

	Linked List	HashMap	Matrix
HeapSpace	10 MB	25 MB	30 MB
Time for Directed search	200ms	100ms	90 ms
Time for Directed search and assigning vehicles	400ms	200ms	156 ms

Table 8: Analysis of the results obtained from the algorithm execution.

Results of cars assigned

Number of cars	5	5	205	205	205	**205**
P or C	P=1.2	P= 1.7	P= 1.1	P= 1.2	P=1.3	**C=6.1 (constant extra time of 6 minutes for all cars)**
Number of routes assigned	2	2	59	57	48	**41**

Table 9: Results obtained from the algorithm execution.

6.Conclusions

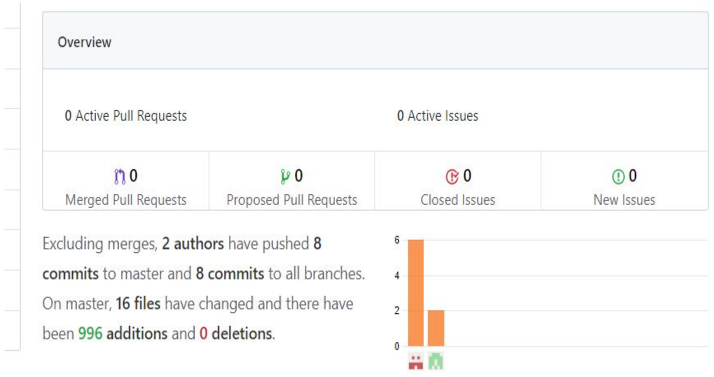
The algorithm proposed in this paper combines different approaches to assign vehicles to shared routes with the intention of setting up a carpooling system. The solution considers a restriction of time, given that a good carpooling system is meant to assign rides without adding too much time to total time of each customer. The first solution proposed used a HashMap as the primary structure, as described in literal 4.1, this structure was

replaced with a matrix of edges which is fitting given that the maximum data given is 205 cars.

This constraint p was initially thought as a constant value that when multiplied to the original time it took for each user to get to the destination would give us the maximum time that user could take getting to the destination in the new assigned routes. When developing the first solution we noticed the p value would result in some users having their maximum time be too large for the real life applications of this problem whereas for some other users the maximum time was so close to the original time that there was virtually no change in the route the car took. These factors made us conclude that a constant value of extra time for all cars would be more fitting , the difference in results can be seen in table 9 .Taking a constant time c of 6 minutes for all cars resulted in 42 routes assigned which is very close to the optimal solution.

The first solution only computed alternative routes to get to the destination picking up other cars (instead of the direct route) without taking into account that after this is process is done , some nodes will be left unassigned, the final solution fixes this problem and assigns those nodes that were not assigned during the directed search. The first solution assigned 76 cars , the second solution assigned 59 and the final solution where p was replaced by c gave 42 cars, which is very close to the optimal 4

ANNEX 1 : Progress in github



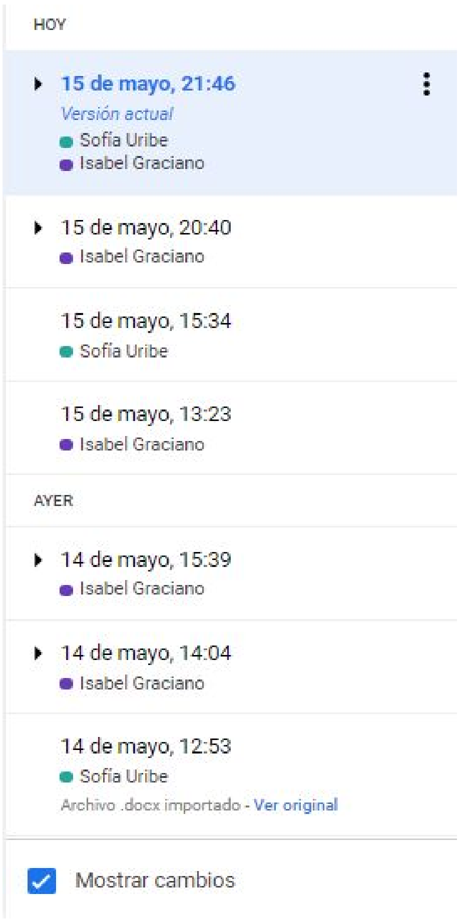
Overview

0 Active Pull Requests | 0 Active Issues

0 Merged Pull Requests | 0 Proposed Pull Requests | 0 Closed Issues | 0 New Issues

Excluding merges, 2 authors have pushed 8 commits to master and 8 commits to all branches. On master, 16 files have changed and there have been 996 additions and 0 deletions.

ANNEX 2:Report progress



HOY

15 de mayo, 21:46
Versión actual
Sofía Uribe
Isabel Graciano

15 de mayo, 20:40
Isabel Graciano

15 de mayo, 15:34
Sofía Uribe

15 de mayo, 13:23
Isabel Graciano

AYER

14 de mayo, 15:39
Isabel Graciano

14 de mayo, 14:04
Isabel Graciano

14 de mayo, 12:53
Sofía Uribe
Archivo .docx importado - Ver original

Mostrar cambios

ANNEX 3: Gradual progress.

TEAM MEMBER	DATE	DONE	DOING	TO DO
ISABEL	29/02/2019	Discussing task distribution		Code
SOFIA	29/02/2019	Discussing task distribution		Code
SOFIA	01/04/2019	Start the code		Code comments and report
ISABEL	23/04/2019	Finalizing the code		Report
SOFIA	23/04/2019	Finalizing the code		
ISABEL	13/05/2019	Add the data structure in the report		
SOFIA	13/05/2019	Add time complexity and memory consumption in the report		
ISABEL	15/05/2019	Do the PowerPoint presentation		
SOFIA	15/05/2019	Analysis of the results, execution times and complexity		
ISABEL	15/05/2019	Upload files in github		

REFERENCES

[1] Activity. Toy problems in real world. Taken from: https://www.computingatschool.org.uk/data/tft/02/04Activity_Toy_Problems_Real_World.pdf.

[2] Fischer, G. and Nakakoji, K. Amplifying designers' creativity with domain-oriented design environments. in Dartnall, T. ed. Artificial Intelligence and Creativity: An Interdisciplinary Approach, Kluwer Academic Publishers, Dordrecht, 1994, 343-364.

[3] Dijkstra's Algorithm -- from Wolfram Math World. (2019). Retrieved from http://mathworld.wolfram.com/DijkstrasAlgorithm.html

[4] Shortest Path Problem -- from Wolfram Math World. (2019). Retrieved from http://mathworld.wolfram.com/ShortestPathProblem.html

[5] Solution Methods for VRP | Vehicle Routing Problem. (2019). Retrieved from http://neo.lcc.uma.es/vrp/solution-methods/

[6] Vehicle Routing Problem | OR-Tools | Google Developers. (2019). Retrieved from https://developers.google.com/optimization/routing/vrp

[7] INRIX Global Traffic Scorecard. (2019). Retrieved from http://inrix.com/scorecard/

[8] Harcourt, P., & Harcourt, P. (2016). Route Optimization Techniques: An Overview, *7*(11), 1367–1372.

[9] Dijkstra Algorithms - an overview | ScienceDirect Topics. (2019). Retrieved from https://www.sciencedirect.com/topics/computer-science/dijkstra-algorithms

[10] Minimum Spanning Trees. (2019). Retrieved from https://algs4.cs.princeton.edu/43mst/

[11] Dréo, J. (2019). [Image]. Retrieved from https://es.m.wikipedia.org/wiki/Archivo:Aco_TSP.svg

[12] University of Sheffield. (2019). *Final Results of Dijkstra's Algorithm* [Image]. Retrieved from https://ptwiddle.github.io/Graph-Theory-Notes/s_graphalgorithms_shortest-paths.html