A Graph Model for a Ridesharing Platform

David D. Silva, Joana T. Ferreira, Luís M. Cunha, and Rosaldo J.F.Rossetti

Faculty of Engineering, University of Porto
Rua Dr Roberto Frias, s/n, 4200-465, Porto, Portugal
{up201705373, up201705722, up201706736, rossetti}@fe.up.pt

Abstract. Commuting is a source of air pollution, traffic and waste of resources, when not optimized. With the rise of Smart Cities, it is important to address this issue, by introducing Intelligent Transport Systems. The aim of this research is to automate and optimize the process of ridesharing among a selected community, by creating a platform where users can be efficiently paired with each other, in order to optimize the cost and time of their commute and reduce their environmental impact and overall traffic congestion of the city.

Keywords: Smart cities, intelligent transportation systems, transportation, ridesharing, graph algorithms, Floyd-Warshall, greedy algorithms.

1 Problem description

Many people have the car as their first option for their daily commute, whose abusive use leads to environmental damage and the depletion of renewable resources. A way of improving this situation, as well as reduce the road traffic is adopting **ridesharing**, a practice in which people who leave in neighboring zones and have common destinations share the same car with the goal of minimizing the number of cars in circulation using only a small portion of their capacity.

This way, the number of cars in circulation is decreased and the cost of the commute may be divided by all passengers, reducing the cost of the commute for those who provide the car and giving a comfortable mean of transportation for the passengers.

The problem may be abstracted as a graph problem, in which the edges represent roads and the vertices represent geographical locations in a map, which may correspond to the origins and destinations of the passengers.

To closely model real life, each passenger will have a minimum pickup time and a maximum arrival time of their commute, which must be considered when a solution of proposed. This approach is the focus of the paper because it models the real-life situation we are trying to solve where people who have a daily commute to the same place (go to the same college or work in the same firm) share a ride.

2 Problem Formalization

2.1 Entry data

- $G_{initial}$ ($V_{initial}$, $E_{initial}$) – Weighted directed graph without negative weights:

. $V_{initial}$ – vertices that represent places in a road map. Each vertex v ∈ $V_{initial}$ contains the following information:

.id – unique integer identifier for each vertex of the graph;

. address – complete address of the place represented by the vertex;

. coords – geographical coordinates of the vertex in the map, being coords.lat the latitude and coords.long the longitude;

. people – list that contains the people present at the location corresponding to the vertex;

. adj – list of adjacent edges. That is, e ∈ v.adj ⇒ e.vi = v

. $E_{initial}$ - directed edges that represent roads or paths between places in the map. Each edge e = (vi, vf) ∈ $E_{initial}$ contains the following information:

. id - unique integer identifier for each edge of the graph;

. vi – origin vertex of the edge;

. vf – destination vertex of the edge;

. maxSpd – maximum allowed speed;

. dist – length of the road/path;

. fTraffic – Traffic factor. Factor that represents the road traffic intensity of the road/path

. time – represents the estimated time to go from the node represented by v_i to v_f . This time can be obtained through a formula dependent from the values of maxSpd, dist, fTraffic.

- People – list of people that need a ride. Each person p ∈ People, contains the following information:

. id - unique integer identifier for each people in the network;

. origin – address corresponding to the point where the person picks up the ride;

. destination – address corresponding to the person's destination;

. minDepTime – minimum time of the day (hour and minute) that the person wishes to pick up the ride;

. maxArrTime - maximum time of the day (hour and minute) that the person wishes to arrive at their destination;

. departureTime – time at which the person picks up the ride (might have an undefined value during the algorithm)

- Cars – List of cars. Each car corresponds to a person that is willing to give a ride to people that are in the graph $G_{initial.}$ One car c ∈ Cars possesses the following information:

. id - unique integer identifier for each car in the network

. idDriver – unique integer identifier of the driver of c

. numMaxPass- maximum number of passengers the c may carry;

. numPass- current number of passengers in c;

-idCar – identifier of the car that is going to perform the commute in the algorithm. As such, $\exists$ c $\in$ Cars: c.id = idCar.

2.2 Result data

- P – list of people (p $\in$ *People)* that participate in the ride.
- S - sequence of vertices (*v* $\in$ $V_{initial}$), sorted by the order they must be travelled in order to go through all origins and destinations of people in P. If S(i) is the i-th element of S, then S(0).address = People[Cars[idCar].idDriver].origin and S(n-1).address = People[Cars[idCar].idDriver].destination, where n is the number of nodes in the path. For example, if S = {v1,v5,v3,v1,v6}, then the car will start at the address corresponding to the vertex v1 and will follow the path represented by the edge (v1,v5) $\in$ $E_{initial}$, next (v5,v3),(v3,v1), and (v1,v6), being v6 the vertex with the address corresponding to the address of the driver.
- S = { } $\Rightarrow$ There isn't a possible path between the driver's origin and its destination;
- P = { } $\Rightarrow$ No ride is given to any people in the network. In this case, S will be the shortest path between the driver's origin and its destination or, as is seen in the previous point, the inexistence of a path between the two points.

2.3 Restrictions of entry data

$\forall$v1, v2 $\in$ $V_{initial:}$ v1$\neq$v2 $\Rightarrow$ v1.id $\neq$ v2.id;
$\forall$v1, v2 $\in$ $V_{initial:}$ v1$\neq$v2 $\Rightarrow$ v1. coords $\neq$ v2.coords $\land$ v1.address < v2.address ;
$\forall$e $\in$ adj: adj $\in$ $V_{initial}$$\Rightarrow$ p $\in$ *People;*
$\forall$e1, e2 $\in$ $E_{initial}$: e1$\neq$e2 $\Rightarrow$ e1.id $\neq$ e2.id ;
$\forall$e $\in$ $E_{initial}$$\Rightarrow$ e.$v_i$$\neq$ e.vf$_;$
$\forall$e $\in$ $E_{initial}$$\Rightarrow$ e.$v_i$$\in$ $V_{initial}$ $\land$ e.vf$\in$ $V_{initial}$;
$\forall$e $\in$ $E_{initial}$$\Rightarrow$e.maxSpd > 0 $\land$ e.dist > 0 $\land$ e.fTraffic > 0 $\land$ e.time > 0 ;
$\forall$e $\in$ $E_{initial}$$\Rightarrow$e.time = f(maxSpd,dist,fTraffic);
$\forall$p $\in$ people$\Rightarrow$ p $\in$ *People;*
$\forall$p $\in$ *People* $\Rightarrow$ $\exists$v1, v2 $\in$ $V_{initial}$: v1.address = p.origin $\land$ v2.morada = p.destination;
$\forall$p $\in$ *People* $\Rightarrow$ p.origin $\neq$ p.destination $\land$ p.minDepTime < p.maxArrTime ;
$\forall$p $\in$ people $\Rightarrow$ p.departureTime $\geq$ p.minDepTime $\land$ p.departureTime < p.minArrTime*;*
$\forall$p1,p2 $\in$ *People* : p1$\neq$p2 $\Rightarrow$ p1.id $\neq$ p2.id ;
$\forall$*c1,c2* $\in$ *Cars* : c1$\neq$c2 $\Rightarrow$ c1.id $\neq$ c2.id ;
$\forall$*c* $\in$ *Cars* $\Rightarrow$ c.numMaxPass $\geq$ 0 $\land$ c.numPass $\geq$ 0 ;
$\forall$*c* $\in$ *Cars* $\Rightarrow$ c.numPass $\leq$ c.numMaxPass ;
$\forall$c $\in$ *Cars* $\Rightarrow$ $\exists$p $\in$ People: c.idDriver = p.id ;

$\exists c \in$ *Cars: c.id = idCar.*

2.4 Restrictions do result data

$\forall p \in P \Rightarrow p \in$ *People;*
$\forall v \in S \Rightarrow v \in V_{initial}$*;*

2.5 Goal function

The solution to the given problem will be to minimize the total time of the ride, as well as guarantee that each person that participates in the ride, including the driver, reaches its destination before the upper limit given by that person (p. maxArrTime).

3 Solution

3.1 Graph pre-processing

Obtaining the simplified network. The pre-processing of the graph as the goal of finding a simplified graph (G_{simple}) of the initial graph ($G_{iniital}$). In the simplified graph only has the information relevant to the algorithms used in the problem's solution, for that, the graph only has the vertices of $G_{initial}$ that have people (origins) or that are destinations of people in the network. The graph also as the information about the shortest distance between all pairs of vertices in $G_{initial}$.

The construction of G_{simple} demands the use of an algorithm of the computation of the "shortest path" between all pairs of vertices. The algorithms considered for the effect are the Floyd-Warshall algorithm, that as a complexity of $\Theta(|V|^3)$, and Dijkstra's algorithm applied once for each vertex (this algorithm will be refered as All-Pair-Dijkstra, or APD, from now on).

The complexity of Dijkstra's algorithm may vary according to the implementation used for the mutable priority queue (MPS) used by the algorithm. Using a common MPS, the complexity of the algorithm is $O((|V| + |E|).\log|V|)$, which applied to all vertices translates to a total complexity of APD algorithm of $O(|V|.(|V| + |E|) \log|V|)$. On the other hand, using a Fibonacci Heap, it's possible do reduce the complexity of Dijkstra's algorithm to $O(|V|.\log|V| + |E|)$ and the total complexity to $O(|V|^2 .\log|V| + |V|.|E|)$.

To choose the appropriate algorithm, it's necessary to do an analysis of $G_{initial}$, specifically, to its density. In fact, if the graph isn't very dense, such that $|E|\sim|V|$, the complexity of the APD algorithm, may be approximated by $O(|V|^2 .\log|V| + |V|^2)$ or just $O(|V|^2.\log|V|)$. However, in the case of a dense graph, in which $|E| \sim |V|^2$, the complexity would be $O(|V|^2 .\log|V| + |V|^3)$. We can conclude that to a sparse network, there are advantages in using Dijkstra’s algorithm and in a dense network it is advantageous to use the Floyd-Warshall’s algorithm.

Another advantage of the FW algorithm is that it produces a matrix containing the distances between all pairs of vertices. This is very useful as it provides a very efficient way of consulting the distances between the vertices in the graph during the algorithms.

Taking the observations made into account, and if the city that is being modelled is dense (worst case) we chose to use Floyd-Warshall’s algorithm.

Simplified Graph Reduction. The next pre-processing phase of the graph consists of doing a selection on the people who are in the network in order to remove people who have incompatibilities with the driver of the car.

The first incompatibility is the existence of people whose maximum arrival time is less than the driver’s departure time.

Other people who are incompatible are the ones whose sum of the minimum departure time with the shortest path between the driver’s origin and the person’s destination (in this study case is equal to the driver’s destination) is larger than it’s maximum arrival time.

The selection may involve the removal of vertices of the simplified graph, because they represent places that have people in need of a ride in it (or their destination) and, in the case all the people in a vertex are removed by the selection, and it isn’t the destination of any other person on the network, the vertex is no longer relevant to G_{simple}.

The previous phase gives also information about the connectivity between elements of the network, that is, about the existence (or not) of a path to the given vertex. As such, an analysis of the distance-matrix generated by the Floyd-Warshall algorithm, specifically the column of the origin vertex allows us to verify if there are nodes in the network that are unreachable from the driver (“infinite” distance in the matrix) and these nodes must be removed.

3.2 Finding the path

To find the path a greedy strategy is applied. Using a priority queue and an associated “compatibility” function, the people that are more compatible with the ride are chosen. The car starts in its origin node and builds a priority queue. It takes from the queue the most compatible person, checks if it’s time restrictions are compliant with every person already in the car(including the driver) and adds it to the ride and moves to the node corresponding to that person. The driver repeats this process until the car is full or until there aren’t any more compatible people in the designated area. The queue gives a priority to its elements according to the function ϕ.

The function $\phi(v)$. In order to minimize the total time of the car ride, a function ϕ is used to give a numeric value to each node $v \in V_{simple}$ that allows to evaluate its compatibility with the ride in question. The function $\phi(v)$ depends on four factors that have different weights in its result. the function is calculated one time per person on the vertex and the result is the max among them.

The first factor is the "time" between the current node of the driver and the node in analysis(v), in a way that the function will privilege people that are near (in a time perspective) the driver. The second factor is the Euclidian distance between the destination of the person in v to the driver's destination. This factor isn't very impactful in the main study case (where every person's destination is the same as the driver's) but it comes in effect in the third case where the driver and its passengers may have different destinations. The third factor is the Euclidian distance between v and the destination of the driver, with goal of, in a similar way as the A* algorithm, give and incentive to the progression towards the direction of the destination node.

The final factor is very important to reduce the hazards of the greedy aspect of the algorithm. When a new person is to be added to the existing ride, it must be compatible to all people already present in the car. As such, when a person is added, all future candidates are now restricted by the time constraints of that person and others already in the car. To avoid choosing people who will restrict greatly future candidates the final factor should be the degree of similarity between the maximum arrival time of the driver and the person to be added.

fillCarGreedy is a greedy algorithm, that for a driver with a specific destination chooses a path and the people to "pickup". The search starts with the driver and, until the car is full or there are no more people that should be added to the ride, chooses the next person to add.

For each choice, a limited part of the graph is explored (the region where the vertices have a distance to the current vertex less than **dis.** Every vertex in that area is added to the priority queue Q, according to the function ϕ.

Implementation. In the next step the highest priority vertex in Q is chosen. If it's possible to go through that vertex and arrive on time at the destination then the person is added to car and the algorithm starts again, exploring that vertex. If it is not possible, the algorithm selects the next person in the queue, repeating the process until a suitable person is found or until the queue is empty. If the queue is empty, then the driver should not pickup anyone else and should go straight to the destination.

The pseudo-code to this algorithm is the following:

```
function fillCarGreedy(Gsimple , car):

    Priority_queue Q
    dis // maximum search distance
    currentVertex ← car.driver
    currentTime ← car.driver.minDepTime
    dest ← car.driver.destination
    maxArrHour ← car.driver.maxArrTime

```

```
10.    while not car.isFull():
11.        Q.makeEmpty()
12.
13.        for each v in Gsimple:
14.          if dist[currentVertex, v] < dis and v.hasPerson and not car.hasPerson(v.Person):
15.                Q ← v
16.          endif
17.        endfor
18.
19.        while Q is not empty:
20.          u ← Q.extractMin()
21.          predictedArrival ← currentTime+ time[currentVertex, u] + time[u, dest]
22.          if predictedArrival< maxArrHour and predictedArrival< u.maxArrTime:
23.            currentTime ← currentTime + time[currentVertex, u]
24.            p ← maxϕPerson(u)
25.            car ← p
26.            currentVertex ← u
27.
28.            if u.maxArrTime < maxArrHour:
29.              maxArrHour ←u.maxArrTime
30.            endif
31.
32.            break
33.          endif
34.        endwhile
35.
36.        if Q is empty:
37.           return car
38.        endif
39.
40.   endwhile
41.
42.   return car
```

The lines 3-8 have the goal of initializing the variables and the priority queue that uses the function ϕ(v) to define the priority. The initial vertex is the driver's vertex, and the ride starts at the start time corresponding to the driver. While the car is not full (car.isFull()), the queue is populated (13-17) and the new element to be part of the ride is chosen (lines 19 to 34). The line 24 is used to disambiguate the result of a vertex that contains more than one person, returning the person that has the highest value for ϕ. Because everyone is going to the same location, there is only a need to assure that it’s possible to arrive at the destination before minimum arrival time of all the passengers and driver.

3.3 Extrapolating result data

In the end of the proposed solutions a vector containing the people that will participate in the ride is returned (P). This vector is ordered by the order in which these people must be added to the ride. To obtain the vector of nodes corresponding to the car’s path, it’s necessary to use G_{simple} and the matrices generated by the FW algorithm to find the sequence of nodes corresponding to the paths (S).

An optimization could be made to S to try to eliminate repetitions that lead to an unnecessary passing through some nodes.

3.4 Balance between optimization and computational cost

To find the solution that is closer to the optimal, it’s possible to adapt these algorithms. For this, instead of considering only the first option of the priority queue, the first n nodes of the queue should be considered. This way, the algorithm may explore more than one path selecting the one with lower cost. Choosing a larger n will yield a solution closer to the optimal one.

However, it should be considered that increasing n implies and polynomial increase in the problem’s complexity. Nonetheless as every combination is independent, using parallel computation technics and depending on the hardware, it is possible to reduce the computation time significantly. On the other side, using dynamic programming to avoid repeating calculations may lead to better performance.

4 Use cases and functionalities

Currently, the accentuated development of the mobile platform yields an ideal base for the implementation of the given ridesharing solution. As such, the use case of the given solution would consist of integrating it in a smartphone app that would allow people would allow people who need rides or people who want to give a ride to minimize environmental costs and travel-costs to register.

The simplification of the graph that is described earlier is not shown to the user and can be maintained through many iterations of the algorithm unless the information about the roads is changed overtime. The changes between iterations are related to the current time and time constraints as well as the people that are looking to receive a ride.

The use of the app considers two cases, the case of a person looking for a ride and a person that, having a car, is looking to give a ride. The people that sign up to give a ride need to give their origin and destination and a minimum departure and maximum arrival time. The driver must give the same information as well as the capacity of the car. The driver is given a path that he must take as well as the people he must pick up in order to arrive at his destination before the specified maximum time.

5 Theoretical analysis of the temporal complexity of the solution

The temporal complexity of the proposed algorithm is composed of two different phases: the graph pre-processing and the algorithm run on the simplified graph.

5.1 Pre-processing

A depth first search (DFS) in the initial graph G1(V1, E1) from the driver's node allows for the removal of the unreachable nodes from the driver. A DFS has a worst-case complexity of O(|V1| + |E1|) and the subsequent construction of the first simplified graph (G2) implies a complexity of O(|V1| + |E1|). Adding the two we obtain a complexity of 2.O(|V1| + |E1|), which can be simplified to O(|V1| + |E1|).

Having the simplified graph G2(V2, E2), a second simplification is made so that the main algorithm can have access to the values of the costs of the shortest paths between all pairs of vertices and remove incompatible people.

For that, the Floy-Warshall algorithm is applied which has a complexity of O(|$V2^3$|). With this information the second simplified graph G3(V3) is built and this graph only has the vertices that are origins of people (including the driver) and their common destination. The graph also holds the adjacency matrix resulting from the FW algorithm. The construction as complexity O(|V2|) because it only implies adding nodes of G2 that have people to G3.

The removal of incompatible people has a complexity of O(|V2|+P), where P is the number of people in the graph, leaving us with a total complexity of O(|V1| + |E1|) + O(|V2|3) + O(|V2|) +O(|V2|+|P|) for the processing phase.

The proportions between the number of nodes in G1, G2 and G3 are dependent on several factors like the origin of the driver, the destination, the amount of people in the initial graph and the dispersion of nodes.

5.2 fillCarGreedy Algorithm

The fillCarGreedy algorithm as a temporal complexity that depends on several factors. In fact, the search for the start and destination vertices of the path yields a linear complexity with the number of vertices in the graph (reminding that in this phase the graph is G3), O(|V3|).

The inside of the while loop of the lines 10-40 consists of the construction of a priority queue with the vertices of the graph and, following that, the extraction of people until a person that is compatible with the ride is found. The construction of the queue implies the insertion of nodes of the graph and the definition of their priority, yielding a complexity of O(log(|V3|). |V3|.O(ϕ)) where O(ϕ) is the complexity of the priority function. This complexity is the worst-case complexity as in reality only vertices inside a certain radius from the current vertex are added.

The extraction of vertices from the queue has a worst-case complexity of O(|V3|.log(|V3|)), being the case where only one person is compatible and it is the one with lowest complexity of all people in the graph(very rare because the priority function as compatibility as the main focus).

Finally, the while loop of lines 10-40 is executed, at most, one time per empty seat in the car, that is a constant value (and usually equal to five) and, as such, can be ignored for the total time complexity of the algorithm.

The fillCarGreedy algorithm has a total complexity of O(|V3|) + O(log(|V3|).|V3|.O(ϕ)) + O(log(|V3|).|V3|), simplified to O(log(|V3|).|V3|.O(ϕ)).

6 Conclusions and Future Work

The purpose of this paper was to create an automatic tool to optimize ridesharing. In order to do this, an algorithm was created to group people together in the same car, according to their departure's location and time restrictions. By running this algorithm for all the cars available, each time removing the selected passengers from the graph, it is possible to find a trip for each user.

Although the results are not optimal for every person in the platform, the current solution does return a logical solution within a reasonable time and space complexity, as discussed in section 5.

The next step would be to implement the platform, find possible optimizations to this algorithm and adapt it in order to accommodate people with different destinations in the same trip.

References

1. "Introduction to Algorithms", 3rd Edition, T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein., MIT Press, 2009;
2. "Slides de Concepção e Análise de Algoritmos", R. Rossetti, L. Ferreira, L. Teófilo, J. Filgueiras, F. Andrade, CAL, MIEIC, FEUP;