Real-Time Carpool Rideshare Optimization

By

Myles Gould and Matthais Nobles

Tables Of Contents

I. Introduction

Ride-sharing has gained significant popularity in recent years as a sustainable and cost-effective mode of transportation. The basis of ride sharing involves individuals sharing a vehicle to travel to a common destination, or similar destination. The concept of ride-sharing has been around for centuries, with people sharing rides to various destinations such as their workplace. However, with the advancement of ride-sharing platforms, such as Uber and Lyft, ride-sharing has become more organized and accessible to a broader audience.

One of the primary benefits of ride-sharing is that it reduces traffic congestion, especially in urban areas where traffic can be seen as a major issue. The cost of congestion in the United States is around $121 billion per year, which includes 5.5 billion hours of time lost to sitting in traffic and an extra 2.9 billion gallons of fuel burned [1]. By sharing rides, fewer cars are on the road, which leads to less congestion and faster travel times. In turn, this can reduce carbon emissions and improve air quality, making ride-sharing a more sustainable transportation option. In addition to reducing traffic congestion, ride-sharing can also help reduce transportation costs for both drivers and passengers. Drivers can save money on gas, while passengers can save on transportation costs by not having to own a car or pay for parking.

Moreover, ride-sharing can also improve accessibility to transportation for people who do not have access to vehicles or public transportation. By sharing a ride, individuals can reach their destination even without owning a car, knowing how to drive, or live in an area where public transportation is not readily available.

However, the task of matching large groups of riders to a fleet of shared vehicles in real time, which is essential for large-scale ride-sharing, requires mathematical models and algorithms that are not adequately addressed by existing solutions. Thus, current models are inefficient and do not fully address the potential of ride-sharing due to limiting the amount of rider requests per vehicle. Therefore, for our project we intend to create a real-time redistributive carpooling ride-sharing model that minimizes congestion and energy consumption. We look to assess the tradeoff between vehicle capacity, rider waiting time, rider travel delay, and operational costs. Thus, we will be able to analyze the effect of carpooling in ride-sharing.

II. Technical Background

A. Matlab

We chose Matlab for developing our algorithm due to its ability to handle complex mathematical computations, advanced optimization algorithms, extensive data analysis, visualization capabilities, and its built-in libraries. Matlab allows us to formulate our optimization problem and then solve it using the same software platform. Matlab has an extensive optimization toolbox. The toolbox includes tools for constraint-based optimization, such as setting upper and lower bounds to our solutions along with equality and inequality constraints for design variables.

This can be useful in developing algorithms that satisfy multiple constraints, such as passenger preferences and vehicle availability. The toolbox provides an easy-to-use interface that allows developers to quickly prototype and solve optimization problems. Matlab also provides several ILP libraries, such as the Integer Programming Solver, which allows for efficient solving of large-scale ILP problems. These libraries implement a range of advanced algorithms, such as branch and bound. Lastly, Matlab provides advanced data visualization and analysis tools, such as plotting and data analysis functions. These tools can be used to analyze large data sets and visualize the results of optimization algorithms, which can be useful in identifying trends and patterns in ride-sharing data. [2]

B. Graphical Methods

Graphical methods can be helpful in testing edges for ride-sharing optimization because they provide a visual representation of the problem and can help identify potential issues or solutions that may not be immediately apparent from a purely mathematical perspective.

In ride-sharing optimization, graphical methods can be used to represent the relationships between different variables, such as the locations of requests and drivers assigned to the request, the availability of vehicles, and the constraints on the optimization model. For example, a graph can be used to visualize the locations of requests and drivers, with edges representing the potential routes between different locations to assess if the model has any potential faults. Graphical methods can also be a useful simple method for finding the edges of a multi-design variable unconstrained optimization problems and testing their edges on the constraints to see if they satisfy all necessary condittions.

C. Integer Linear Programming

Integer Linear Programming (ILP) is a mathematical optimization technique that is commonly used in building algorithms for ride-sharing optimization because it allows for efficient and effective allocation of resources, such as vehicles and drivers, to serve a large number of ride-share requests with different origins and destinations while restricting all design variables to be integers..

In a ride-sharing optimization problem, the goal is to assign requests to available vehicles in a way that minimizes the overall cost, such as the total travel time or the number of vehicles required. ILP can help find the optimal solution by modeling the problem as a set of linear equations and inequalities that capture the constraints and objectives of the problem. For example, the ILP model can include variables representing the assignment of each ride-share request to a vehicle, as well as constraints to assess if a request has found a trip to be included on to assure that a driver gets assigned the maximum of one trip. The objective function can then be defined to minimize the energy consumption and traffic congestion in the model by minimizing the total delay time and the number of requests that have found a trip. By using ILP, the algorithm can quickly and accurately allocate resources to serve a large number of ride-share requests in real-time, leading to increased efficiency and customer satisfaction.

III. Optimization Formulation and Analysis Model

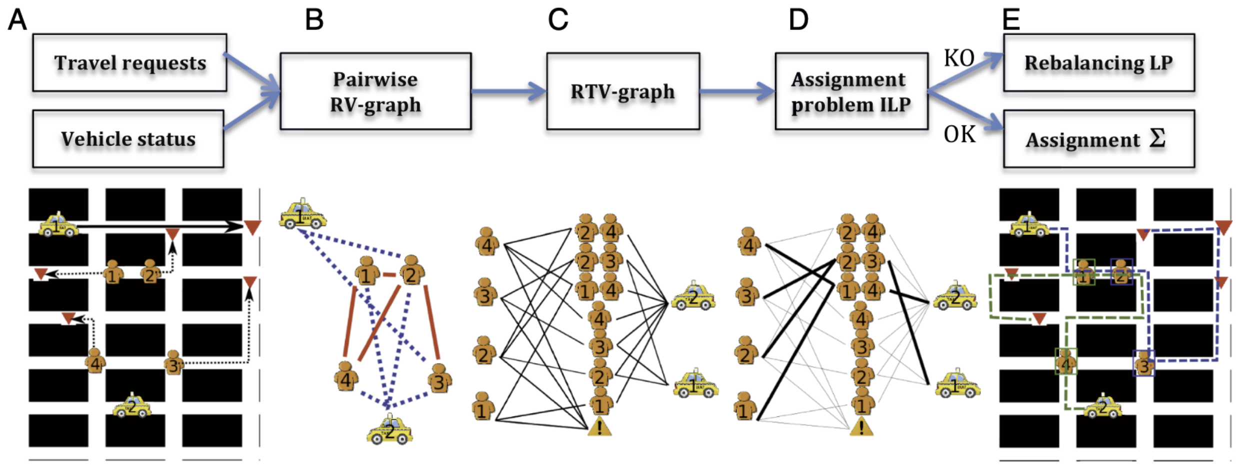


Figure 1: Diagram depicting the optimization model

The figure above [1] is a step-by-step schematic representation of how our algorithm operates by taking a cluster of requests and matching them to multiple vehicles. The algorithm we created was inspired by the research article, "On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment" [1]. Our algorithm goes through a series of steps, ultimately leading to the integer linear programming which provides optimal assignment, by assigning vehicles to requests. All steps for this algorithm were created using MatLab.

The design variables of our experiment are $e_{i,j}$ and x_k, which are both binary. The variable $e_{i,j}$ represents a trip set, i, being carried out by vehicle, j. If a feasible trip will be carried out by vehicle j, $e_{i,j} = 1$, whereas if the trip won't occur by vehicle j, then $e_{i,j} = 0$. The variable x_k represents if a request can be met. For example if request 1, r_i, can't be satisfied then $x_1 = 0$, however if r_1, can be satisfied then $x_1 = 1$. The total number of design variables of our experiment also varied as new requests were made or if requests were completed. For example at t=0, if there are two vehicles and one request, r_1, assuming that all constraints are satisfied, then there are two possible trips, e_1 and e_2, as both vehicles can each complete r_1. There is only one x value, x_1, as there is only one request, making the design variables $[e_1, e_2, x_1]$.

Now, at t = 1, if another request, r_2, is made and assuming that each vehicle can only complete each request individually, then the design variables are [e_1, e_2, e_3, e_4, x_1,x_2], as each vehicle can satisfy each request individually, therefore increasing the number of feasible trips. The number of x variables also increases as another request has been made. The constraints of our design problem were also different for various steps. The primary constraints for step B and C include, max wait time, max total delay time, and max capacity of each vehicle. While, the primary constraints for step D were that each vehicle can be assigned a maximum of one trip and that a request is either carried out or will not be carried out. Constraints will be discussed in further detail later in this section.

Step 1 illustrates a street network that includes four requests denoted by the four orange human icons, which represents the origin of the request, and red triangle icons, representing the customer's desired destination. There are two vehicles represented by yellow car icons. In this diagram, vehicle 1 is carrying one passenger, with the black arrow representing vehicle 1's current trip and the red triangle representing the requested end destination of the passenger. Vehicle 2 is empty. This step, seen in Appendix A section A, focuses on data collection. In this step our model receives the initial starting point of each active vehicle, the pickup point and the requested end point of each request made,each vehicle's speed, the maximum capacity of each vehicle, along with the maximum total delay time and the maximum wait time that each request is willing to make. Each position in our model is identified using an x-y coordinate position. For example a starting point of {0,1} means a request would like to be picked up where x = 0 and y = 1. A request is stored in a 2D matrix-cell where the first index of the cell holds the starting point of a request and the second index holds the requests potential end destination. The maximum wait time is defined as the longest amount of time a request is willing to wait in order to receive a request. If a request can't receive a request until after his max wait time has passed then he will cancel his request and theoretically will find a new form of transportation to his end destination. Wait time,w, is defined as

$$(w) = t_r^p - t_r^r \tag{1}$$

where t_r^p is the time that the requestt is pickup and t_r^r is the time at which the request is made. Max total delay time of a request is the maximum amount of in-vehicle delay plus wait time that a request is willing to wait. In-vehicle delay refers to additional time added to a request's trip if the vehicle picks up another rider before dropping off the request at its end destination. For example, If a vehicle picks up a request and drops the rider off next then the in-vehicle delay is 0. However if a vehicle picks up a request then picks up a second request before dropping off the first request then in-vehicle delay will occur. Total delay time, (δ_r) , is mathematically defined as

$$(\delta_r) = t_r^d - t_r^* \tag{2}$$

where t_r^d is the dropoff time and t_r^* is the quickest time it would take to get to the destination for a request if there is no wait time or in-vehicle delay, or and the shortest path from the origin of a request and the destination of a request. In our model a change of $\Delta x = 1$ or $\Delta y = 1$ occurs in one minute, meaning a vehicle traveling from point {0,1} to {1,1} or vice versace takes one minute. Therefore , t_r^* is defined as

$$t_r^* = t_r + (|x_{end} - x_{start}| + |y_{end} - y_{start}|) \tag{3}$$

Once all of this information was collected we then proceeded to step 2.

Step 2, with code found in Appendix A Section B, shows a pairwise shareability RV-graph to identify the requests and vehicles. The graph is created by representing the request and vehicles as nodes and their pairwise shareability as edges. The links between the cars and people denote potential trips. During this trip we test if it is feasible for each vehicle to pick each individual request up and drop each individual request off with no additional requests being added to the trip, while satisfying each request's maximum wait time and maximum total delay time. If it is feasible for a vehicle to drop a request off, while satisfying the constraints, then a node is created linking the vehicle with the requests as a potential candidate to drop the request off. This was done in MatLab, using a (v x r) matrix called v2rlinks, where j, represents the number of vehicles, and r represents the number of requests. If vehicle, i, can drop off request, j, then v2rlinks(i, j) is set to 1 representing a node between the vehicle and request. If vehicle, i, can't drop off request, j, then v2rlinks(i, j) is set to 0 representing that there is no node connection between the vehicle and request. Once all potential links were created we proceeded to step 3.

Step 3, with code found in Appendix A Section C, shows an RTV-graph to determine the candidate trips and the vehicles that are capable of executing them. The RTV-graph includes nodes representing the requests, vehicles, and potential trips, with edges connecting them based on their compatibility. In the case where a case cannot be satisfied, a yellow triangle node is added to represent this unsatisfied request in the graph. This step is done first by finding all combinations, using comb2.m found in Appendix B, of all potential trips that each vehicle can make given these linkages stored in v2rlinks, but only constrained to their capacity. For example, If vehicle 1 is linked to request 1, r_1, request 2, r_2, and request 4, r_4, with a maximum capacity of two then the potential trips for vehicle 1 are $\{r_1\}$,$\{r_2\}$, $\{r_4\}$, $\{r_1, r_2\}$, $\{r_1, r_4\}$, and $\{r_2, r_4\}$. Once all possible sets of trips are found we test if these trips are feasible given the max wait time and max total delay time requested by each request. This is done by testing these constraints on every permutable pickup, drop off order in which a trip can be carried out until we find if the trip is feasible. For example two permutations of the set $\{r_1, r_4\}$ are $\{r_{1start}, r_{4start}, r_{1end}, r_{4end}\}$ and $\{r_{4start}, r_{1start}, r_{1end}, r_{4end}\}$. If a trip is feasible, which can be tested using feasible.m found in Appendix C, can be carried out by vehicle j we then create a link between the trip set and vehicle j. Once all permuted feasible trip sets are found we look to minimize the cost,which was done using min_cost.m found in Appendix D, of the trip. The cost of the trip is defined as the sum of the total delay times of each request in the set. Changing the order of pickup and drop off of requests can lead to either an increase or decrease in the total cost of the trip due to a request experiencing a long wait time or long in-vehicle delay due to a vehicle taking a longer and inefficient route. For example, if vehicle 1, v_1, starts at $\{1,0\}$, r_1 start point is at $\{2,0\}$ and and r_2 start point is at $\{3,0\}$ and both requests would like to be dropped of at $\{5,0\}$ to minimize total delay, v_1 would pick up r_1 first then would pick up r_2, and would then finally drop both requests off at $\{5,0\}$, thus minimizing the total delay time as opposed to picked up r_2 then turning around to pick up r_1, and then finally dropping both requests off, causing both requests to have a larger total delay time. Once the minimum cost of a feasible trip was found a (e x 3) matrix was created, where e represents the sum of feasible trips that can be carried out by all the vehicles. The first column would share the vehicle that can carry out the trip, the second column would store the optimized order that the trip should be carried out in, and the third column stores the minimum cost of the trip. Once the minimum cost of each feasible trip is calculated then we move along to Step 4.

Step 4, with code found in Appendix A Section D, illustrates an optimal assignment obtained by the ILP solution where vehicles are assigned to requests and asked to carry out their assigned requests. In this step we look to minimize our objective function by using Linear Integer Programming. This step assigns vehicles to trips by looking for the most optimal solution of our objective function, f.m which can be found in Appendix E and shown below.

$$\text{Min } f(x) = \sum c_{i,j}e_{i,j} + \sum c_{\kappa o}x_k \tag{4}$$

The final objective function value, of formula 4 , represents the amount of street congestion and energy consumption that will occur in a scenario and produces a dimensionless value. A larger objective function value means that there is more traffic and more energy consumption will most likely occur while a lower value means less potential congestion and energy consumption. In formula 4, $c_{i,j}$ is the minimum cost of a trip, e_i when carried out by a vehicle j. As previously explained it is the sum of delays (δ_r) that each request will experience if trip $e_{i,j}$ occurs. The longer a trip or the more inefficient a trip the larger the value the objective function will return meaning more potential energy consumption and road congestion. The variable $c_{\kappa o}$ is a large constant set to 50 for unassigned requests used to deter the model from leaving a request unassigned. The purpose of a large c_{ko} constant is to have the model favor a request being part of a trip rather than having the request be unsatisfied. If a request is satisfied this would lead to carpooling which is more environment friendly, causing less energy consumption and congestion, thus a lower objective function value. However, if a request is not satisfied this could possibly result in a rider using another ride-share service for an individual ride or leading to a rider driving to their destination, thus increasing the objective function value as this would lead to more total energy consumption and traffic. Therefore we need to deter the model from leaving a request unsatisfied as if a request is unsatisfied, this could potentially increase street congestion and energy consumption even further. Using LIP on MatLab we were able to limit all variables to integers. However, LIP doesn't limit variables to binary so the upper bound for each design variable needed to be set to 1 and the lower bound of each design variable needed to be set to 0. Our initial conditions for our optimization was to set all trips, e_i equal to 0 and all x_k variables equal to 1 meaning the initial conditions where all trips don't occur and all requests are not satisfied. This was done for simplicity as the number of design variables could change throughout iterations. Lastly our objective function had two constraints, with code for both constraints seen in Appendix F. The first contraints, shown below in formula 5, assumes that each vehicle is assigned to one trip or zero trips.

$$(\sum e_{i,j}) \leq 1 \quad \text{for all vehicles } j \tag{5}$$

Formula 5 states that the sum of the binary values of every trip, e_i, that can be carried out by j must be less equal to one. Since all values are binary this ensures that each vehicle receives a maximum of one trip. This constraint is created for each vehicle. Our second constraint, shown below in formula 6, assures that no more than two trips containing request k occur and that if no trip containing request carrying request k occurs then request k is unsatisfied and $x_k = 1$.

$$(\sum e_{i,j}) + x_k = 1 \quad \text{for all requests k} \qquad (6)$$

Further, this formula enforces that the sum of all trips containing request k must be equal to one and $x_k = 0$, meaning a trip containing request k occurs and request k is satisfied, or the sum of all trips containing request k equal to zero and $x_k = 1$, meaning a trip containing doesn't occur and request k request k isn't satisfied.

An easier mathematical representation of our objective function and its constraints to follow can be shown below in Figure 2.

$$\begin{array}{lll}
\textbf{2:} & \Sigma_{optim} := \arg\min_{\mathcal{X}} \sum_{i,j\in\mathcal{E}_{TV}} c_{i,j}\epsilon_{i,j} + \sum_{k\in\{1,\dots,n\}} c_{ko}\chi_k & \\
\textbf{3:} & \textbf{s.t.} \sum_{i\in\mathcal{I}^T_{V=j}} \epsilon_{i,j} \leq 1 & \forall v_j \in \mathcal{V} \\
\textbf{4:} & \sum_{i\in\mathcal{I}^T_{R=k}} \sum_{j\in\mathcal{I}^V_{T=i}} \epsilon_{i,j} + \chi_k = 1 & \forall \mathrm{r}_k \in \mathcal{R}
\end{array}$$

Figure 2: Optimization function and constraints on the optimization

Lastly, Step 5, with code found in Appendix A Section D, depicts the planned route for the two vehicles. In this step we look to carry out each optimal assignment by having the vehicles carry out their assigned trip until a new request is made. This was done by updating the position of vehicles along with active trip status of each vehicle using the function update_sys.m, shown in Appendix G. This function updated the position of each vehicles, what step of the trip a vehicle is, the amount of passengers a vehicle currently contains, which requests the vehicle has as passengers, which requests that a vehicle has already been dropped off to its end destination, which requests the still need to be picked up by a vehicle, and stores the travel history of each vehicle. This step would occur regardless of if a new vehicle was added to continuously update the system.

If a new request is made as time increases, steps 1, 2, 3, and 4 were then repeated for a new optimal solution, while prioritizing the drop off of passengers currently in the car, by adding them to all potential trips of the vehicle that they are in. Requests not in a vehicle, can then be reassigned to a new trip of a new vehicle to optimize the new system that is at hand.

IV. Analysis of Results

Our results were all analyzed by MATLAB plots and by analyzing final values produced through MATLAB's ILP optimization. The results section will contain data from three different trials run along with analysis on how max capacity, max wait time, max total delay time, the number of vehicles, and the number of requests all affected each of our results, rideshare optimizations and our model.

One of the first trials was a simple problem run to confirm what we were doing was accurate and logical.This was a standard problem where all requests are made at t=0. Request, r_1 start point is at [0,0] with an end destination of [1,1], request, r_2 start point is at [0,1] with an end destination of [1,2], request, r_2 start point is at [0,1] with an end destination of [1,2], r_3 start point is at [3,2] with an end destination of [3,5],and r_4 start point is at [3,3] with an end destination of [3,5]. The capacity of every vehicle was set to two and the maximum wait time was set to 4 and the maximum total delay time was set to 6. Using a simple model allowed us to verify the logistics of our model by confirming calculations made by our model by hand. This allowed us to gain confidence that optimal trips were being assigned, vehicles weren't taking extra long routes and that non-feasible trips were not passing as feasible trips. It also allowed us to gain a better visualization of each step in our model. Figure 3 below shows the path of the two vehicles in our first trial.

.

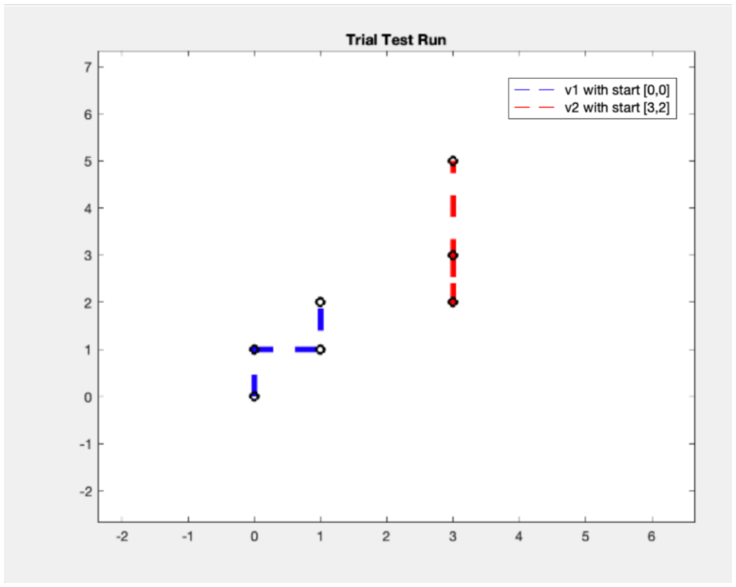
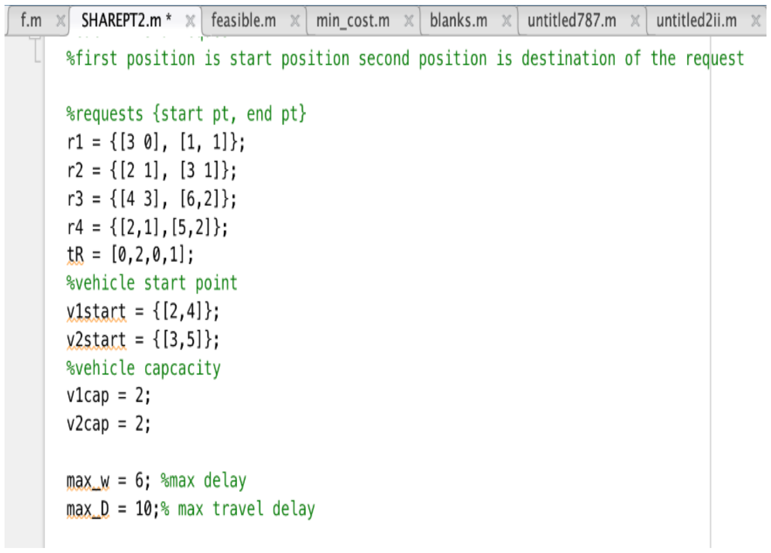


Figure 3: The paths of the two vehicles in the test trial with four ride-sharing requests

These results were identical to the hand calculations that we made and gave us further assurance that our model was accurate, producing a minimum objective function value of 16 with no non-feasible trips being shown as feasible and all total costs being calculated appropriately.

Next we looked at a more complex example and analyzed the effects of tradeoffs and constraints on our system.



f.m SHAREPT2.m * feasible.m min_cost.m blanks.m untitled787.m untitled2ii.m

```
%first position is start position second position is destination of the request

%requests {start pt, end pt}
r1 = {[3 0], [1, 1]};
r2 = {[2 1], [3 1]};
r3 = {[4 3], [6,2]};
r4 = {[2,1],[5,2]};
tR = [0,2,0,1];
%vehicle start point
v1start = {[2,4]};
v2start = {[3,5]};
%vehicle capcacity
v1cap = 2;
v2cap = 2;

max_w = 6; %max delay
max_D = 10;% max travel delay
```

Figure 4: Initial Parameters for trial 1

Figure 4 shows the initial constraints and variables of trial 2. Each r_k represents a request with a start and end points. tR represents the request time of each request with the first index being the request time of r_1 the second index being the request time of r_2, and so on and so forth. Next, you can see the start point of each vehicle as well as its capacity, and finally you can see the universal maximum wait time and maximum total delay time used for all four requests.

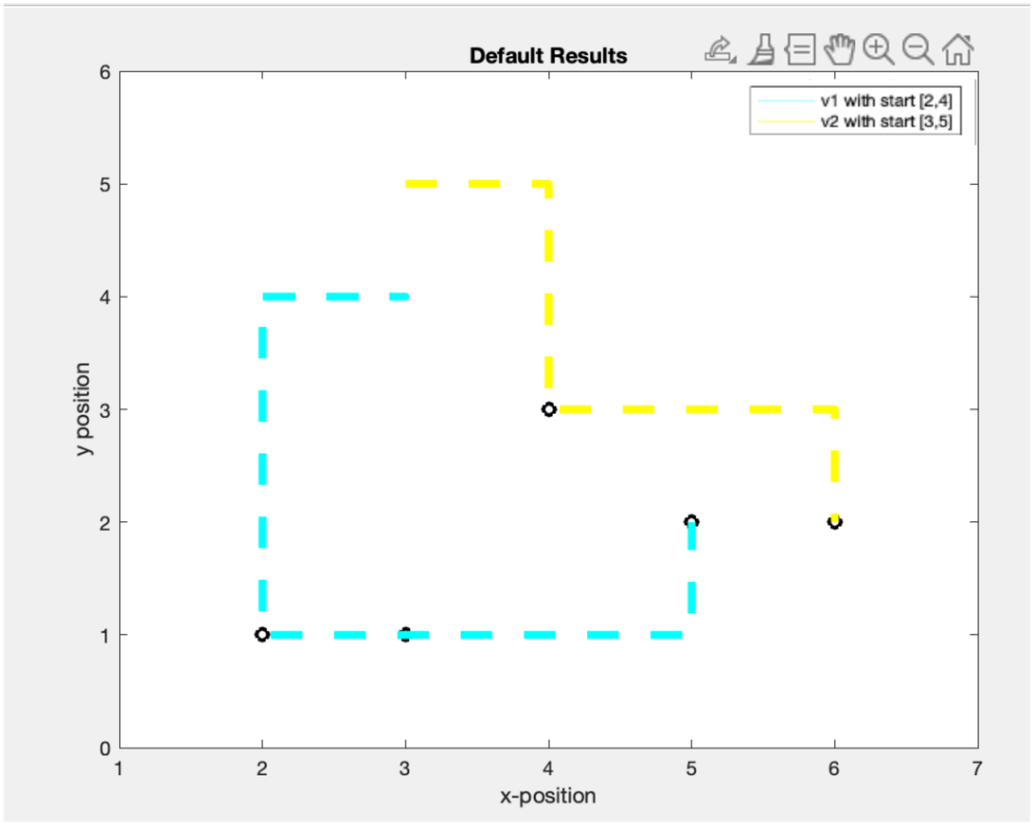


Figure 5: The paths of the two vehicles in trial 1 with four ride-sharing requests

After running our model with the initial parameters, in Figure 5 you can see the path that each vehicle took throughout the whole optimization process. Vehicle 1's path is depicted in blue and vehicle 2's path is depicted in yellow. At t = 0 to t = 1, vehicle 1 goes to serve request r_1, moving from its starting point to point [3,4], and vehicle 2 goes to serve r_3 closer moving towards r_3. However, at t = 1, a new request is introduced, r_4, leading vehicle 1 to fulfill the new request. Vehicle 1 then begins to move towards r_4's origin. This suggests that vehicle 1 couldn't pick up both r_1 and r_3 and got reassigned from satisfying r_1 to satisfying r_3. This shows r_3's trip total cost would be less and the same amount of requests in the total system would be unsatisfied. Vehicle 1 and vehicle 2 both would be unable to satisfy two requests. If vehicle 1 stuck to picking up r_3, its total cost would be larger. At t = 2, a new request also occurs, leading to a new optimal assignment and vehicle 1 being assigned to r_2 and r_4 and r_3 remaining unassigned. No further optimization is needed as no new requests are introduced into the system and all requests are dropped off at their end point.

Next we changed the values of different constraints to see how that would affect the total objective function and the amount of riders satisfied.

Table 1: Comparing the objective cost of different cases

Case Number	Condition	Requests Satisfied	Total Objective Cost
1	Default	3	68

2	Max_wait = 14
	Max_delay = 14
		4	32

3	capacity = 1
	Vehicles = 3
		3	64

4	Vehicles = 3	4	22

Our default condition, case 1, is referring to the initial constraints and conditions seen in Figure 3. During this case, one request was not satisfied, leading to a total objective cost function of 68. When we increased both the maximum wait and maximum delay time of each request we found that all requests were able to be satisfied, leading to a total objective value of 32. This makes sense logistically as the longer people are willing to wait the easier it is to find them and an efficient ride. We also saw that when we only increased the number of vehicles to three they all 4 requests were able to be satisfied with an objective function value of 22, however when we kept the same amount of vehicles and lowered the capacity from two to one only three of the requests with a total objective function value of 64. This suggests that carpooling is an effective method to minimizing street congestion and energy consumption as when we increase the amount of requests that can be satisfied by a vehicle at once, we found the objective function value to be lower, hence less congestion and energy consumed.

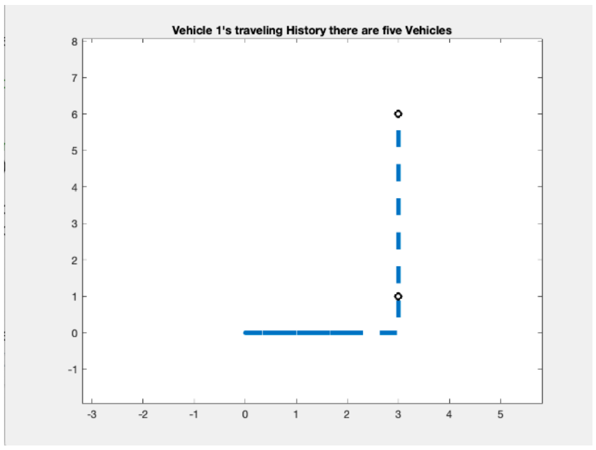


Figure 7: The path of the vehicle in trial 5 with 13 ride-share requests

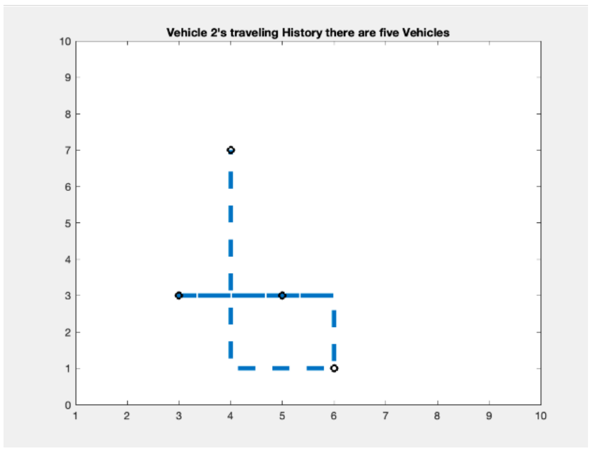


Figure 8: The path of the vehicle 2 in trial 5 with 13 ride-share requests

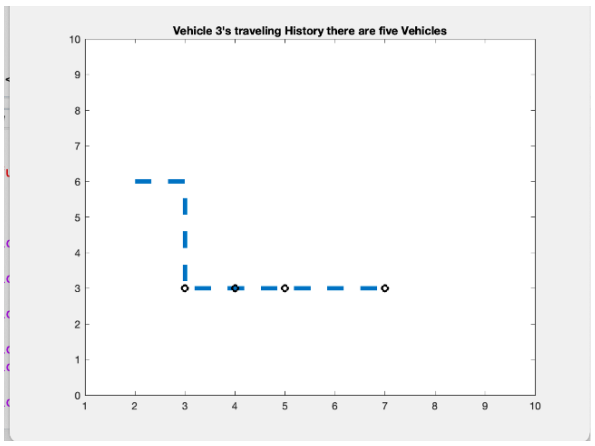


Figure 9: The path of the vehicle 3 in trial 5 with 13 ride-share requests

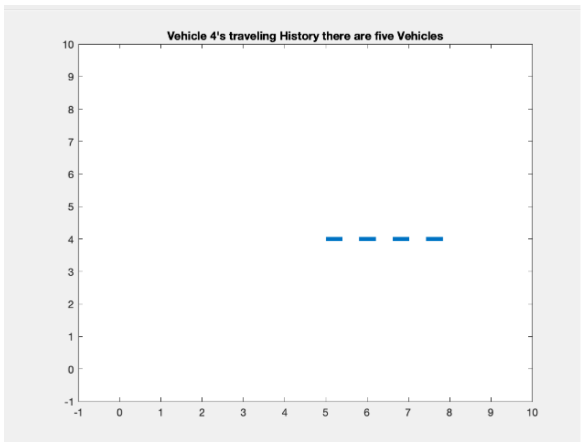


Figure 10: The path of the vehicle 4 in trial 5 with 13 ride-share requests

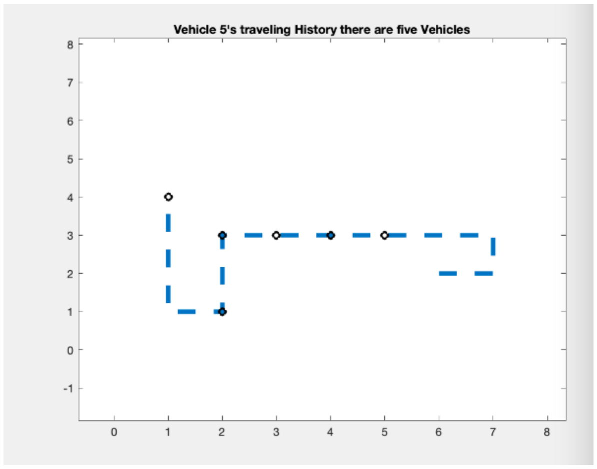
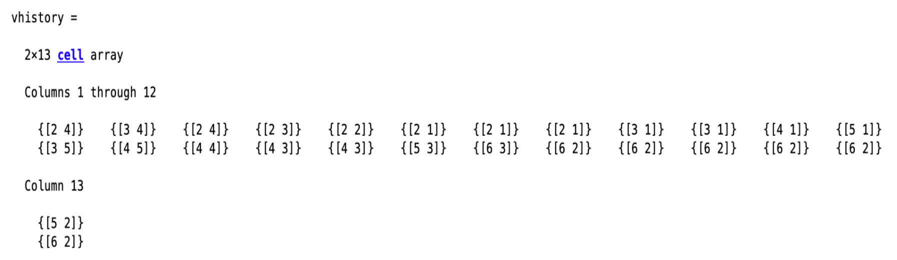


Figure 11: The path of vehicle 5 in trial 5 with 13 ride-share requests

In the final trial, initial values shown in Appendix H , we expanded the scope by increasing the number of vehicles to 5 and the number of ride-share requests to 13. We also increased the maximum travel delay to 14 minutes and the maximum wait time to 13 minutes. Vehicle 1 began at [0,0], then drove to pick up request 3 (r_3) at [3,1] and dropped them off at [3,6]. This path is shown in figure 7. Vehicle 2 began at [3,3] and picked up r_2 at [3,3]. They then drove to [5,3] where they dropped off r_2. Next, they drove to [6,1] to pick up r_{12} and finished by dropping them off at [4,7]. This path is shown in figure 8. Vehicle 3 began at [2,6], then picked up r_{10} at [3,3] then proceeded to pick up r_4 at [4,3]. The vehicle then dropped r_4 at [5,3] and finished the trip by dropping off r_{10} next at [7,3]. This path is shown in figure 9. Vehicle 4 began at [5,4] and drove to [8,4] before stopping and not responding to any requests. This path is shown in figure 10. Vehicle 5 began at [6,2] and first responded to r_5 at [5,3]. Next, it picked up r_8 at [4,3] and proceeded to drop r_8 off at [3,3] and then r_5 at [2,3]. Lastly, it responded to r_{11} at [2,1] and finished the trip by dropping them off at [1,4]. This path is shown in figure 11.

Overall, this trial returned interesting results and shows question as to how large of a scale our model can handle It struggled to minimize the objective function with a resulting value of 184, much higher than the rest of our trials partially due to there being more requests to fulfill, but this could also signify that an improvement in our algorithm is necessary. Vehicle Vehicle 4 had a capacity of 1 making it harder for it to be assigned to a trip to optimize the objective function. Given that vehicle 4 did not complete any request and vehicle 1 only completed 1 request, this negatively impacted the trial the most and future suggests that the number of vehicles need to be added to the objective function to assume that a vehicle is needed and they won't be driving with nobody to pick up or to drop off. Another alternative could be a weighted function to favor filling vehicles of smaller capacity first redistributing trips as it will be easier for larger capacity vehicles to take on more trips at once in the future. Leaving lower capacity vehicles without a direct assignment of a request could lead to a larger objective function in the future as vehicles with larger capacity that are capable of fulfilling a new request could be full and smaller capacity vehicles, such as vehicle 1, are just wasting time traveling. This may further suggest that more improvement in our algorithm is necessary as despite vehicle 4 being next to the start point of multiple requests it received no trips while vehicle 5 was able to satisfy a plurality of tests mainly due to its large capacity. Thus, it is clear that our model is not yet ready for larger scale problems yet. We need to work on improving our method so that it is more efficient and reliable for a bigger scope, however based on our findings we are confident that carpooling ride-sharing can be beneficial in creating a less traffic congested and sustainable environment.



```
vhistory =

  2×13 cell array

  Columns 1 through 12

    {[2 4]}    {[3 4]}    {[2 4]}    {[2 3]}    {[2 2]}    {[2 1]}    {[2 1]}    {[2 1]}    {[3 1]}    {[3 1]}    {[4 1]}    {[5 1]}
    {[3 5]}    {[4 5]}    {[4 4]}    {[4 3]}    {[4 3]}    {[5 3]}    {[6 3]}    {[6 2]}    {[6 2]}    {[6 2]}    {[6 2]}    {[6 2]}

  Column 13

    {[5 2]}
    {[6 2]}
```

Figure 6: Vehicle travel history of vehicle 1 and 2

Lastly, we also collected data on each vehicle's travel history, an example seen in Figure 6, as we believed that it would make sense to add a vehicle competent to the objective function in the future and make each vehicle a design variable. If a vehicle is only able to feasible deliver one vehicle then it could be doing a disservice to the environment and should be considered as such in a future model as it would be the same as if the requester himself drove to his final destination. There is also a possibility that a requester could take the train or walk to his final destination, so assigning only one request to a vehicle could end up being worse than not assigning a vehicle to a request. Also a vehicle is also constantly aimlessly driving without an active feasible ride request then this is leading contributing to unwanted energy consumption and street congestion which negates the purpose of our model

We suggest in the future a variable v_j which represents if a vehicle is necessary to minimize congestion and energy consumption and new objective function:

$$\text{Min } f(x) = \sum c_{i,j} e_{i,j} + \sum c_{\kappa o} x_k + \sum((c_{\kappa o} + 5)/ \sum \#j)\, v_j \qquad (7)$$

where j is the number of requests that would be fulfilled by vehicle j, if we decide to . If v is necessary it equals 1 and if v is not necessary v is zero. This makes adding a vehicle only better than an unassigned trip if it received two or more feasible requests in the trip it is assigned.

V. Verification and Validation

Our model isn't sophisticated or sizable enough to be compared with results of existing rideshare algorithms or models however, we are confident that our model confidently showed the positive effect of carpooling accurately and accurately represented the logistics of the ride sharing algorithm based on our formulation and the results of our experiment.

Our model was based off of an existing study on carpooling that had the capability to run 5,000 or more vehicles. The max wait time in all our trials were set less than the max delay time as the max delay time is the max wait time plus the in-vehicle delay time. First our algorithm needed to know each rider's start and end point along with its max destination, max delay and max wait time before assigning trips . This is similar to ride sharing models such as Uber and Lyft. Before both rider sharing companies can assign you a driver they ask for your pickup location and your drop-off destination along with your preference wait time. Since these companies are mainly focused on linking one request to a vehicle they don't ask you for your max delay time as it will be the same as your request time. After you enter your information these ride sharing algorithms look to find drivers that can satisfy your constraints while completing trips that are in the process and send the request to a driver to optimize their objective function which is primarily focused on maximizing profit and customers. Our algorithm does the same, as it finds all possible combinations of trips while primarily focusing on trips of requests that are already in their vehicle and also attempting to satisfy the max amount of requests.

Once our algorithm completes these steps it then focuses on trying to minimize our objective function. The value of our objective function is dimensionless, but is meant to represent the amount of street congestion and energy consumption caused by its vehicle assignment. The longer a trip the larger its total cost leading to a larger objective function value. This is realistic as a longer trip also leads to an increase in pollution and street congestion as the vehicle continues to use gas and is on the road for a longer period of time. The cost of a request not satisfied was also set to be much larger that any possible total cost that a ride can have. This was to deter the algorithm from leaving a ride unsatisfied due to the possibility of the total cost of the trip being too large as if a vehicle takes on one more request the total cost of its trip will increase, but if no vehicles take the request this could potentially lead to even more pollution or congestion as the request can then go to uber for a ride which then adds another vehicle on the road, leading to more street congestion and more energy vehicle consumption. The cost of an unassigned request is so large as it is preferable for a vehicle to take on an additional passenger than it is for it to not take the request and ride then requests a driver on another app leading to more congestion.

The two constraints for our objective function as shown in formula 5 and 6 allow our algorithm to accurately approach the problem. With the bounds of our design variables being {0,1}, the first constraint shown in 5 assures that each vehicle is only assigned to one trip max or no trips at all. Since each trip, a set of requests. is continuous and with future ride requests being added to the current trip that is possibly already containing passengers, it is not possible for a driver to have two separate trips. The second constraint shows in formula 6 assures that a trip containing a request is carried out and if not then that request is not carried out. This relates to real life situations as if you request an Uber ride they either find you a trip and if they can't then your request is not carried out.

Lastly, the results of our model also lead us to believe that our model properly captured the logistics and physics of ridesharing accurately and effectively on a smaller scope. In the results,we saw that when there was an unsatisfied request and we increased the wait time and delay time of requests, the objective function lowered as all requests were able to be satisfied. This makes sense as the longer a person is willing to wait for the vehicle to pick them up, the easier it is to find a rider for them while still satisfying other requests. We also saw that when there was an unsatisfied request and we increased the number of vehicles, the objective function value decreased as the system was able to satisfy all requests. This also makes sense as the more vehicles the more total potential trips the system can carry out, also making it easier for the algorithm to pair a request to a vehicle. Finally, when we kept three vehicles, but decreased the capacity of each vehicle we found that we went back to only being able to satisfy 3 of 4 requests. This makes sense physically, as if a rider is only able to complete one request at a time it makes it harder for all requests to be carried out.

All these methods of formulation of our optimization and observations that occurred from the results of our algorithm, assure us that our model properly solved ride-sharing from a new numerical sense and the physics of our problem were properly captured.

VI. Conclusion and Future Work

Ride-sharing has become an increasingly popular and sustainable mode of transportation, providing benefits such as reduced traffic congestion. However, current models for large scale ride-sharing have limitations in matching riders to shared vehicles in real-time. In our project, we developed a real time redistributive ride-sharing model using integer linear programming within Matlab which aims to minimize wait times and travel delays by finding the ideal vehicle capacity. While the scope of our model is very small, focusing on just four ride-share requests and two vehicles, we were still able to analyze the effect of ride-sharing and its potential.

In developing the code for this project we had the chance to learn about integer linear programming and its power to provide optimal solutions for complex problems in an efficient manner. We also gained technical insight into the inner workings of ride-sharing services that run systems like Uber and Lyft. Even though the scope of our model was miniscule in the grand scheme of things, we gained an understanding of the complexity for this problem. In regards to our own algorithm, we found the capacity constraint was the most impactful in minimizing the objective function. The higher capacity of the vehicle, the more requests a single vehicle can serve. As a result, the number of cars needed significantly decreases. Of course, this depends on the potential passenger's willingness to wait slightly longer and have a slightly later arrival time. However, if this were widely adapted, the benefit would be less cars on the road and result in less travel delays en route to a passenger’s destination. Therefore, the extra time required when traveling in a high capacity vehicle may balance itself out in the long run.

There are several factors we need to consider in the future to improve our algorithm and make it more representative. Most importantly, this model will need to be scaled up. This can be done by adding significantly more vehicles and ride-share requests, as well as testing more edge cases, as even when five vehicles were added in our model it showed some unwanted behavior in vehicles. Another thing we need to consider is adding an option for a single request to contain multiple riders. Also, a future objective function should include the vehicles. If a vehicle is only assigned to one rider, then that doesn’t improve the environmental energy consumption or street congestion. That could ultimately increase both factors as the purpose of carpooling is to assign a vehicle to multiple riders. Next, factors that impact travel time such as traffic lights, weather and accidents should be integrated into the algorithm. Additionally, more freedom needs to be given to the potential passengers and drivers. Currently, customers are not able to cancel requests in our algorithm and drivers do not have the option to deny requests, both of which happen very frequently. Lastly, there are hundreds of lines of code for our small scale version. We need to find ways to make the algorithm more efficient as a whole. Overall, our model worked well for the small scope and performed ideally in every scenario tested except when there were 5 vehicles and 13 ride-share requests. Thus, we need to do a substantial amount of further testing. This will help us determine which parameters cause our algorithm to fail and allow us to improve upon those aspects to prevent this.

VII. References

[1] J.Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, D. Rus, “On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment”, *PNAS,* 3-Jan-2017. [Online]. Available: https://www.pnas.org/doi/10.1073/pnas.1611675114#body-ref-r1. [Accessed: 16-Mar-2023].

[2] “Optimization Toolbox”, *Mathworks*. [Online]. Available: https://www.mathworks.com/products/optimization.html. [Accessed: 16-Mar-2023].

VIII. Appendix

Appendix A: Main Code File (RideShare.m)

A. Step 1

```
clc;
clear;
%first position is start position second position is destination of the request
%requests {start pt, end pt}
r1 = {[3 0], [1, 1]};
r2 = {[2 1], [3 1]};
r3 = {[4 3], [6,2]};
r4 = {[2,1],[5,2]};
%vehicle start point
v1start = {[2,4]};
v2start = {[3,5]};
%vehicle capcacity
v1cap = 2;
v2cap = 2;
max_w = 6; %max delay
max_D = 14;% max travel delay
%all the requests
r = [];
allr = [r1;r2;r3;r4];
%stores current requests and current wait times of vechiles that are active
curr_R = [];
curr_Mwait =[];
%start point for each vechile
v1start = {[2,4]};
v2start = {[3,5]};
v3start = {[4,1]};
vStart   = [v1start,v2start,v3start];
%stores traveling history of vechiles
vhistory = cell(length(vStart),1);
%vechile speed
vspeed = [1,1,1];
%current position of each vechile
vpos = [v1start, v2start,v3start];
Vdests = cell(length(vStart),length(r));
%capacity of each vechile
vcap = ones(1,3);
%matrix to store passengers of each vechile
Vpass = zeros(length(vcap),length(r));
%matrix to store completed trips
Comp_pass = zeros(length(vcap),length(r));
%max wait times
```

```
max_w1 =6; % t1(pickup) - t1(request)
max_w2 = 6; % t2(pickup) - t2(request)
max_w3 = 6; % t3(pickup) - t3(request)
max_w4 = 6;  % t4(pickup) - t4(request)
max_w = [max_w1,max_w2,max_w3,max_w4];
% max travel delay
max_D = 10;
tStop = 1; %time it takes indivudal to get in car
tR = [0,2,0,1]; % request times
tStar = zeros(1,4); %fastest possible time a rider can get to its destination

Comp_pass = zeros(length(vcap),length(r));
%not used right now
v1Trips = {};
v1Pass = []; %passengers
v1Req = []; %requests
v2Pass = [];
v2Req = [];
%max wait times
max_w1 =6; % t1(pickup) - t1(request)
max_w2 = 6; % t2(pickup) - t2(request)
max_w3 = 6; % t3(pickup) - t3(request)
max_w4 = 6;  % t4(pickup) - t4(request)
max_w = [max_w1,max_w2,max_w3,max_w4];
% max travel delay
max_D = 10;
tStop = 1; %time it takes indivudall to get in car
tR = [0,2,0,1]; % request times

tStar = zeros(1,4); %fastest possible time a rider can get to its destination
```

B. Step 2

```
%% Step 2
tP = @(r) sum(abs(r{2} - r{1})); %time it takes to travel a path


index = 1;
%calculates the fastest possible time a rider can get to its destination for
each request
for i = 1:length(tR)
   tStar(i) = tR(i) + tP(allr(i,:));
End

curr_R = []; %active requests
curr_tR = []; %active requests times
```

```
curr_tStar = []; %fastest possible time a rider can get to its destination for
each requests
%starts at t=0
t = 0;
%iterates through time until a request is made
while isempty(curr_R) == 1
   for p = 1:length(allr)
       if tR(p) == t
           r = [r; allr(p,:)];
           curr_R = [curr_R; allr(p,:)];
           curr_tR = [curr_tR, tR(p)];
           curr_tStar = [curr_tStar, tStar(p)];
           curr_Mwait = [curr_Mwait, max_w(p)];
       end
   end
   if isempty(curr_R) == 1
       t = t + 1;
   end
End

%t initial is the time when the optimization begins to run

t_initial = t;

%stores the number of trips completed
curr_RComp = 0;
lr = size(curr_R,1);


% t isnt t_initial and if there are no active trips left or potential trips
left then optimization ends

while (t == t_initial) || (curr_RComp <
(sum(Vpass,'all')+sum(v2rlinks,'all'))|| tripsleft > 0)
   for ll = 1:size(vhistory,1)
       vhistory{ll,(t+1)} = vpos{ll};
   end
   for ww = 1:length(r)
       if(t  - curr_tR(ww) > curr_Mwait(ww))
           curr_RComp = curr_RComp+1;
           curr_R{ww,1} = [];
           curr_R{ww,2} = [];
       end
   End

%%% Following Code was used to input requests manually instead of automatically


%     req = input("do you want to make a request?  ");
%     if(isa(req, 'char'))
%         if (isequal(lower(req), 'yes' ) || (lower(req) ==  'y' ))
%             st = input('Where would you liked to be picked up from?   ');
%             if(isa(st, 'double')&& (length(st) ==2))
%                 nd = input('Where would you liked to be dropped off?   ');
%                 if(isa(nd, 'double') && (length(nd) ==2))
%                     wt = input('how long are you willing to wait?   ');
%                     if(isa(wt, 'double') && length(wt) ==1)
%                         tm = input('what time would you like to be picked up?
');
%                         if(isa(tm, 'int')) && (tm < t)
%                             allr = [allr, {st,nd}];
%                             curr_Mwait = [curr_Mwait,wt];
%                             tR = [tR,tm];
%                             tStr = tm + tP({st,nd});
%                             tStar = [tStar,tStr];
%                         else
%                             fprintf("\nREQUEST WAS NOT MADE.\nReminder: The
time you request must be equal to or after the current time\n\n")
%
%                         end
%                     else
%                         fprintf("\nREQUEST WAS NOT MADE.\nReminder: The wait
time must be a single integer value\n\n")
%                     end
%                 else
%                     fprintf("\nREQUEST WAS NOT MADE.\nReminder: The droppoff
destination must contain two integer values a\n\n")
%                 end
%             else
%                 fprintf("\nREQUEST WAS NOT MADE.\nReminder: The pickup
destination must contain two integer values a\n\n")
%             end
%         else
%             fprintf("\nREQUEST WAS NOT MADE.\nReminder: type yes or y to
request a ride\n\n")
%         end
%     else
%         fprintf("\nREQUEST WAS NOT MADE.\nReminder: you must type characters
a\n\n")
%     end
   if t ~=0
       for p = 1:length(allr)
           if tR(p) == t
               r = [r; allr(p,:)];
               curr_R = [curr_R; allr(p,:)];
               curr_tR = [curr_tR, tR(p)];
               curr_tStar = [curr_tStar, tStar(p)];
               curr_Mwait = [curr_Mwait, max_w(p)];
```

```
            end
        end
    end
run = 0;
if (lr ~= size(curr_R,1)) ||  ( t==0)
   lr = size(curr_R,1);
   run = 1;


%vpos = [vpos{1}+vspeed(1)*t,vpos{2}+vspeed(2)*t];
v2rlinks = zeros(length(vcap),length(curr_R));
%tests  if a vechile can satsifty each rrequest by itself
for jj = 1:length(vcap)
   for kk = 1:length(curr_R)
       if isempty(curr_R{kk,1}) == 0
       wait_kk = sum(abs(vpos{jj}-curr_R{kk,1}));
       if curr_Mwait(kk) >= wait_kk
           Kk_drop = tStop + wait_kk + curr_tR(kk) +
sum(abs(curr_R{kk,2}-curr_R{kk,1}));
           Kd = Kk_drop - curr_tStar(kk);
           if (max_D >= Kd)
               v2rlinks(jj,kk) = 1;
           end
       end
       end
   end
end
%%check if there is a link between the v and the r
```

C. Step C

```
%% Step 3
sz = 0;
%checks for the max combination of rides
vec = [];
for a = 1:length(curr_R)
   if (isempty(curr_R{a,1}) == 0)
       vec = [vec,a];
   end
end

for vv = 1:length(vec)
   combinations = combntns(vec,vv);
   sz = sz + size(combinations,1);
end
riders = 0;
%matrix for all possible trips
if sz  == 0
```

```
   Ptrips =cell(length(vcap),1);
   for jjj = 1:size(updated_trips,1)
   end
else
   Ptrips =cell(length(vcap),sz);
end
%vector for trips that are actually feasible
%each vechile has its owns row
trips = {};
for a = 1:length(vcap)
   vP = find(v2rlinks(a,:)); %finds were there is a link between a vechile and
a request
   if isempty(vP) == 0
       Ptrip= comb2(vP,vcap(a)); %makes all possible combinations out of the
requests with a capcity contstraint
       for gg = 1:length(Ptrip)
           Ptrips{a,gg} = Ptrip{gg}; %adds it to a matrix
       end
   end
end
v2rlinks = zeros(length(vcap),length(curr_R));
%iterate through Ptrips vector
Ts = 0;
for aa = 1:size(Ptrips,1)
   index = 1;
   for bb = 1:size(Ptrips,2)
       ord = Ptrips{aa,bb};%takes vector of a trip
       if sum(Vpass(aa,:)) > 0
           passengers = find(Vpass(aa,:)== 1);
           if isempty(ord) %if there is no trip in the index then the length is
zero
               per = perms(passengers); %%creates all possible orders that
riders can be picked up or dropped off in a ride
               per = unique(per,'rows');
               disp(per);
               for cc = 1:size(per,1)
               %feasible function tests if a trip i feasible
                   if
((feasible(vpos(aa),per(cc,:),curr_Mwait,curr_tStar,max_D, tStop,r,
curr_tR,t,Vpass(aa,:))) == 1)
                       trips{aa,index} = per(cc,:); %adds it to trips that are
feasible
                       index = index + 1;
                       Ts = Ts + 1;
                       break
                   end
               end
               break
           end
```

```
            per = perms([ord,ord,passengers]); %%creates all possible orders
that riders can be picked up or dropped off in a ride
            per = unique(per,'rows');
        else
            per = perms([ord,ord]); %%creates all possible orders that riders
can be picked up or dropped off in a ride
            per = unique(per,'rows'); %%makes sure there arent duplicate rows
        end
        disp(per)
        if isempty(ord) %if there is no trip in the index then the length is
zero
           break
        end
         for cc = 1:size(per,1)
            %feasible function tests if a trip i feasible
           if sum(Vpass(aa,:) ) >0
                if ((feasible(vpos(aa),per(cc,:),curr_Mwait,curr_tStar,max_D,
tStop,r, curr_tR,t,Vpass(aa,:))) == 1)
                    trips{aa,index} = per(cc,:); %adds it to trips that are
feasible
                    index = index + 1;
                    Ts = Ts + 1;
                    break
                end
           elseif((feasible(vpos(aa),per(cc,:),curr_Mwait,curr_tStar,max_D,
tStop,r, curr_tR,t,[])) == 1)
                trips{aa,index} = per(cc,:); %adds it to trips that are feasible
                index = index + 1;
                Ts = Ts + 1;
                break

           end
        end
    end
end


%% Step 4
%create matrix
%matrix stores every trip
%first col is the vechile of the trip second col is requests in the trip
%and last row is the trips total cost
matrix = cell(Ts,3);
index = 1;
for v = 1:size(trips,1)
   for vv = 1:size(trips,2)
       if trips{v,vv} == 0
           break
       end
       %adds vechile to matrix

       %finds the min cost that a trip can have
       if sum(Vpass(v,:))>0
           [i3,i2] = min_cost(vpos(v), trips{v,vv},curr_tStar,tStop,r
,curr_tR,t,Vpass(v,:));
       else
           [i3,i2] = min_cost(vpos(v), trips{v,vv},curr_tStar,tStop,r
,curr_tR,t,[]);
       end
       if isempty(i2) == 0
       matrix{index,1} = v; %adds vechile
       matrix{index,2} = i2; %adds order of the min cost trip
       matrix{index,3} = i3; %add cost of trip
       index = index + 1;
       end
   end
End
```

D. Step D

```
len_e = size(matrix,1); %number of trips
len_x = length(r); %number of requests
intcon = 1:(len_x + len_e); %all values must be ints
lb = zeros(1,len_x + len_e); %creates lower bound of zero
ub = ones(1,len_x + len_e); %creates upper bound of 1
x0 =[zeros(1,len_e), ones(1,len_x)]; %intial values are all requests are not
satisfied


for s = 1:size(Comp_pass,1)
   if (sum(Comp_pass(s,:))>0)
       for ss = 1:size(Comp_pass,2)
           if (Comp_pass(s,ss) == 1)
               for sss = 1:size(matrix,1)
                   if (matrix{sss,1} == s)
                       matrix{sss,2} = [ss,matrix{sss,2}];
                   end
               end
           end
       end
   end
end
   [A,b,Aeq, beq,Vs] = nonclon2(len_e, len_x,matrix,trips);
   y = f(len_e,len_x, matrix); %creates objective function
   options = optimoptions('intlinprog','Display', 'iter');
   [x,fval,exitflag, output] = intlinprog(y,intcon, A, b, Aeq, beq, lb, [],
x0,options);
end



VasTrips = {}; %x matrix used to plug into a function to update the system

index = 1;
if t == 0
for xx = 1:len_e
   if (x(xx) == 1)
       VasTrips{index,1} = matrix{xx,1};
       VasTrips{index,2} = matrix{xx,2}; %TRIP
       len_trip = length(matrix{(xx),2});
       VasTrips{index,3} = zeros(1,len_trip); %tracks what part of trip we in
       VasTrips{index,4} = zeros(1,length(curr_tR)); %check if person is picked
up and rdrop off
       VasTrips{index,5} = 0;  %did u just stop
       index = 1 + index;
   end
end
else
   for xx = 1:len_e
       if (x(xx) == 1)

       VasTrips{index,1} = matrix{xx,1};
       passengers = find(Vpass(matrix{xx,1},:) == 1);
       Comppassengers = find(Comp_pass(matrix{xx,1},:) == 1);
       if (isempty(Comppassengers)) == 0 && (run==1)
           VasTrips{index,2} = [Comppassengers,passengers,matrix{xx,2}]; %TRIP
       elseif (run == 1)
           VasTrips{index,2} = [passengers,matrix{xx,2}];
       else
           VasTrips{index,2} = matrix{xx,2};
       end
       len_comp = length(find(Vpass(matrix{xx,1},:) == 1)) +
2*length(find(Comp_pass(matrix{xx,1},:) == 1)) ;

       if matrix{xx,1} >size(updated_trips,1)
       for oo = 1:size(updated_trips,1)
           if updated_trips{oo,1} == matrix{xx,1}
               status = updated_trips{oo,4};
           end
       end
       else
           status = updated_trips{matrix{xx,1},4};
       end
       one1 = find(status == 1);
       VasTrips{index,4} = zeros(1,length(curr_tR));
       for xxx = 1:length(one1)
           VasTrips{index,4}(one1(xxx)) = 1;
       end
       two2 = find(status == 2);
       for xxx = 1:length(two2)
           VasTrips{index,4}(two2(xxx)) = 2;
       end
       len_trip = length(VasTrips{index,2}) ;
       VasTrips{index,3} = [zeros(1,len_trip)];
       VasTrips{index,3}(1:len_comp) = 1;
         %did u just stop
       index = 1 + index;
       end
   end
end
   t = t + 1;
  [vpos, updated_trips,new_cap,Vdests,Vpass,vcap,
r,curr_RComp,curr_R,Comp_pass] = update_sys(vpos,vspeed, VasTrips, r,
t,vcap,Vdests,Vpass, curr_RComp,curr_R,Comp_pass);
  tripsleft = 0;
  for yy = 1:size(updated_trips,1)
      for yyy = 1:length(updated_trips{yy,3})
          if updated_trips{yy,3}(yyy) == 0
              tripsleft = tripsleft +1;
          end
      end
  end
   %update position and status of trip
disp(updated_trips)
disp(VasTrips)
end
```

Appendix B: Finds all possible combinations (comb2.m)

```
%used to find all possible combinations of feasible requests

function orders = comb2(indexes,vcap) %add capacity
orders = {};
index = 1;
for i = 1:vcap
   %creates all possible combinations given a contraint
   %the max being in a trip is vcap
   com = combntns(indexes,i);
   for ii = 1:size(com,1)
       orders{index} = com(ii,:);
       index = index + 1;
   end

end
end
```

Appendix C: Tests feasibility of trip (feasible.m)

```
%tests if a combination of feasible requests of a vehicle is possible

function b = feasible(Vstart, requests,wait_max,tStar,max_D, tStop,r
,tR,t_now,Vpass)
%returns 1 if trip is feasible and returns 0 if trip isnt feasible
b=1;
x = 1;
pickedup = zeros(1,length(wait_max)); %0 is untouched %1 is pickedup %2 is
dropped off
%only checked pickups and not drop offs
%check when request is different
if sum(Vpass) > 0
   for ii = 1:length(Vpass)
       if Vpass(ii) == 1
           pickedup(ii) = 1;
       end
   end
end
while x <= length(requests)
   if (x ==1) && (pickedup(requests(x))==0)
       %first iteration it tests the time a person has to wait
       wait = sum(abs(Vstart{1}-r{requests(x),1}));
       if (wait == 0)
           t_now = wait + tR(requests(x))+t_now+tStop;
       else
           t_now = wait + tR(requests(x))+t_now;
       end
       if wait_max(requests(x)) < wait %if the wait is bigger than the max wait
the trip isnt feasible
           b=0;
           break

       end

       pickedup(requests(x)) = pickedup(requests(x)) + 1; %adds 1 to signify
the request has been picked up
       x = x+1;
   elseif (x == 1) && (pickedup(requests(x))==1)
       t_dropoff = sum(abs(Vstart{1}-r{requests(x),2})) + t_now;
       tD = t_dropoff - tStar(requests(x));
       if (max_D < tD)
               b = 0; %not feasible
               break
       end
       t_now = t_dropoff;
       pickedup(requests(x)) = pickedup(requests(x)) + 1;
       x = x+1;
```

```
    else
        if pickedup(requests(x)) == 0 %if index hasnt been pickeed up
            if pickedup(requests(x-1)) == 1 %and previous player has been
dropped off
                tarrive = t_now + sum(abs(r{requests(x),1}- r{requests(x-1),1}))
+ tStop;
                wait = max(0,(tarrive-tR(requests(x))));
            else %previous player has been ggotten picked up
                tarrive = t_now + sum(abs(r{requests(x),1}- r{requests(x-1),2}))
+ tStop;
                wait = max(0,(tarrive-tR(requests(x))));
            end
            if wait_max(requests(x)) < wait
                b=0;
                break
            end
            if (tarrive < tR(requests(x))) %if the driver arrives before the
request
                t_now = tR(requests(x));
            else            %driver arrives at time of request or after
                t_now = tarrive;
            end


        end
        if pickedup(requests(x)) == 1 %if driver now needs to be dropped ff up
            if pickedup(requests(x-1)) == 1 %pervious rider was dropped off
                t_drop = sum(abs(r{requests(x),2}- r{requests(x-1),1})) + tStop
+ t_now;
            else %pervious rider was picked up
                t_drop = sum(abs(r{requests(x),2}- r{requests(x-1),2})) + tStop
+ t_now;
            end
            %finds its total delay and tests if it is larger than the max
            %delay
            tD = t_drop - tStar(requests(x));
            if (max_D < tD)
                b = 0; %not feasible
                break
            end
            t_now = t_drop;
        end
        pickedup(requests(x)) = pickedup(requests(x)) + 1; %update status of the
request
        x = x + 1;

    end
```

```
end
end
```

Appendix D: Minimize Cost of Trip(min_cost.m)

```
%minimizes cost of the trip

function[min_tD, best_order] = min_cost(Vstart, T,tStar,tStop,r
,tR,t_now,Vpass)
   pOrders = perms(T);
   pOrders = unique(pOrders,'rows');
   d_pickedup = zeros(1,max(T));
   rt_now = t_now;
if sum(Vpass) > 0
    for ii = 1:size(Vpass,2)
         if Vpass(ii) == 1
              d_pickedup(ii) = 1;
         end
    end
end
   for i = 1:size(pOrders,1)
       order = pOrders(i,:);
       disp(order)
       total_tD = 0;
       pickedup = d_pickedup;
       x=1;
       t_now = rt_now;


       while x <= length(order)
           if (x ==1) && (pickedup(order(x))==0)
               wait = sum(abs(Vstart{1}-r{order(x),1}));
               if (wait == 0)
                   t_now = wait + tR(order(x))+t_now+tStop;
               else
                   t_now = wait + tR(order(x))+t_now;
               end

           elseif (x == 1) && (pickedup(order(x))==1)
               t_dropoff = sum(abs(Vstart{1}-r{order(x),2})) + t_now;
               tD = t_dropoff - tStar(order(x));
               total_tD = tD + total_tD;
               t_now = t_dropoff;

           else
               if pickedup(order(x)) == 0
                   if pickedup(order(x-1)) == 1
                       tarrive = t_now + sum(abs(r{order(x),1}-
r{order(x-1),1})) + tStop;

                                            else
```

```
                        tarrive = t_now + sum(abs(r{order(x),1}-
r{order(x-1),2})) + tStop;

                                           end
                    if (tarrive < tR(order(x)))
                       t_now = tR(order(x));
                    else
                        t_now = tarrive;
                    end


                end
                if pickedup(order(x)) == 1
                    if pickedup(order(x-1)) == 1
                        t_drop = sum(abs(r{order(x),2}- r{order(x-1),1})) +
tStop + t_now;
                    else
                        t_drop = sum(abs(r{order(x),2}- r{order(x-1),2})) +
tStop + t_now;
                    end
                    tD = t_drop - tStar(order(x));
                    total_tD = tD + total_tD;
                    t_now = t_drop;
                end

            end
            pickedup(order(x)) = pickedup(order(x)) + 1;
            x = x + 1;
        end
        if i == 1
           best_order = order;
           min_tD = total_tD;
        elseif total_tD < min_tD
           best_order = order;
           min_tD = total_tD;
        end
        fprintf(" order is \n")
       % disp(order)
        %disp(total_tD)



    end
end
```

Appendix E: objective function(f.m)

```
function [y]=f(len_e,len_x, matrix)
y = [];
cO = 50;
for i = 1:len_e
   y = [y;matrix{i,3}];
end
for j = 1:len_x
   y = [y;cO];
end
```

Appendix F: constraints(nonclon2.m)

```
function [A,b,Aeq, beq, Vs] = nonclon2(len_e, len_x,matrix,trips)
A = zeros(size(trips,1), (len_e+len_x));
Aeq = zeros(len_x, (len_e+len_x));
b = ones(size(A,1),1);
Vs = cell(1, size(trips,1));
for i = 1:size(matrix,1)
   v = matrix{i,1};
   A( v,i) = 1;
end
index = 1;
for ii = 1:len_x
   for iii = 1:size(matrix,1)
       if (sum(find(matrix{iii,2}==ii))>0)
           Aeq(ii,iii) = 1;
       end
   end
   Aeq(ii, (len_e+ii))= 1;

end
beq = ones(size(Aeq,1),1);
end
```

Appendix G: updates the system everytime new request is added(update_sys.m)

```
%updates the status of each request, wheter they are a passenger of a vehicle
if their trip has been compleed or etc

% also updated the position and capacity of each vehicle in the system along
with stores how far in the trip that a vehicle is

function [vpos, VasTrips, vcap,vdest,
vpass,vcap1,curr_R,curr_RComp,curr_R2,Comp_pass] = update_sys(vpos,vspeed,
VasTrips,curr_R,t,vcap,vdest, vpass,curr_RComp1,curr_R2,Comp_pass)
for index = 1:size(VasTrips,1)
   Ctrip = VasTrips{index,3};
   trip =  VasTrips{index,2};
   happened = VasTrips{index,4};
   v = VasTrips{index,1};
       for i = 1:length(Ctrip)
           if(Ctrip(i)==0)
               next = trip(i);
               disp(vpos{index})
               if (t == 1) && (vpos{index}(1) == curr_R{next,1}(1)) &&
(vpos{index}(2) == curr_R{next,1}(2))
                   Ctrip(i) = 1;
                   happened(next) = 1;
                   vcap(v) = vcap(v) -1;
                   vpass(index,next) = 1;
                   vdest{index,next} = curr_R{next,1};
                   curr_R2{next,1} = [];
                   curr_R2{next,2} = [];
                   vcap1 = vcap;
                   break

               end

               if happened(next) == 0
                   if (vpos{index}(1) >  curr_R{next,1}(1))
                       vpos{index}(1) = vpos{index}(1)  - vspeed(index);
                   elseif (vpos{index}(1) <  curr_R{next,1}(1))
                       vpos{index}(1) = vpos{index}(1) + vspeed(index);
                   else
                       if (vpos{index}(2) >  curr_R{next,1}(2))
                           vpos{index}(2) = vpos{index}(2)  - vspeed(index);
                       elseif (vpos{index}(2) <  curr_R{next,1}(2))
                           vpos{index}(2) = vpos{index}(2) + vspeed(index);
                       else
                           Ctrip(i) = 1;
                           happened(next) = 1;
                           vcap(v) = vcap(v) -1;
                           vpass(index,next) = 1;
```

```
                            vdest{index,next} = curr_R{next,1};
                            curr_R2{next,1} = [];
                            curr_R2{next,2} = [];
                            vcap1 = vcap;
                        end
                    end
                elseif happened(next) == 1
                    if (vpos{index}(1) >  curr_R{next,2}(1))
                        vpos{index}(1) = vpos{index}(1)  - vspeed(index);
                    elseif (vpos{index}(1) <  curr_R{next,2}(1))
                        vpos{index}(1) = vpos{index}(1) + vspeed(index);
                    else
                        if (vpos{index}(2) >  curr_R{next,2}(2))
                            vpos{index}(2) = vpos{index}(2)  - vspeed(index);
                        elseif (vpos{index}(2) <  curr_R{next,2}(2))
                            vpos{index}(2) = vpos{index}(2) + vspeed(index);
                        else
                            happened(next) = 2;
                            Ctrip(i) = 1;
                            vcap(v) = vcap(v) + 1;
                            curr_RComp1 = curr_RComp1 + 1;
                            vcap1 = vcap;
                            vpass(index,next) = 0;
                            Comp_pass(index,next) = 1;
                            vdest{index,next} = [];


                        en
                    end
                end
                break
            end
        end
    VasTrips{index,3} = Ctrip;
    VasTrips{index,2} = trip;
    VasTrips{index,4} = happened;

end
curr_RComp = curr_RComp1;
vcap1 = vcap;
end
```

Appendix H: Initial Conditions for five vehicles and three requests

```
%requests {start pt, end pt}

r1 = {[0, 0], [1, 1]};

r2 = {[3, 3], [5,3]};

r3 = {[3,1], [3,6]};

r4 = {[4,3],[5,3]};

r5 = {[5, 3], [2, 3]};

r6 = {[6, 3], [3,3]};

r7 = {[0,1], [1,4]};

r8 = {[4,3],[3,3]};

r9 = {[7, 4], [10,6]};

r10 = {[3, 3], [7,3]};

r11 = {[2,1], [1,4]};

r12 = {[6,1],[4,7]};

r13 = {[7, 4], [2,6]};

max_w = 6; %max delay

max_D = 14;% max travel delay

%all the requests

r = [];

allr = [r1;r2;r3;r4;r5;r6;r7;r8;r9;r10;r11;r12;r13];

%stores current requests and current wait times of vechiles that are active

curr_R = [];

curr_Mwait =[];

%start point for each vechile

v1start = {[0,0]};

v2start = {[3,3]};

v3start = {[2,6]};

v4start = {[5,4]};

v5start = {[6,2]};
```

```
vStart   = [v1start,v2start,v3start, v4start,v5start];

%stores traveling history of vechiles

vhistory = cell(length(vStart),1);

%vechile speed

vspeed = [1,1,1,1,1];

%current position of each vechile

vpos = [v1start, v2start,v3start, v4start,v5start];

Vdests = cell(length(vStart),length(r));

%capacity of each vechile

vcap = [2,3, 2, 1, 3];

%matrix to store passengers of each vechile

Vpass = zeros(length(vcap),length(r));

%matrix to store completed trips

Comp_pass = zeros(length(vcap),length(r));

%max wait times

max_w1 =13; % t1(pickup) - t1(request)

max_w2 = 13; % t2(pickup) - t2(request)

max_w3 = 13; % t3(pickup) - t3(request)

max_w4 = 13;  % t4(pickup) - t4(request)

max_w = [max_w1,max_w2,max_w3,max_w4,13,13,13,13,13,13,13, 13,13];

% max travel delay

max_D = 20;

tStop = 1; %time it takes indivudal to get in car

tR = [2,1 , 4, 5, 3, 0, 3 , 2, 1,  0, 5, 4, 2]; % request times
```