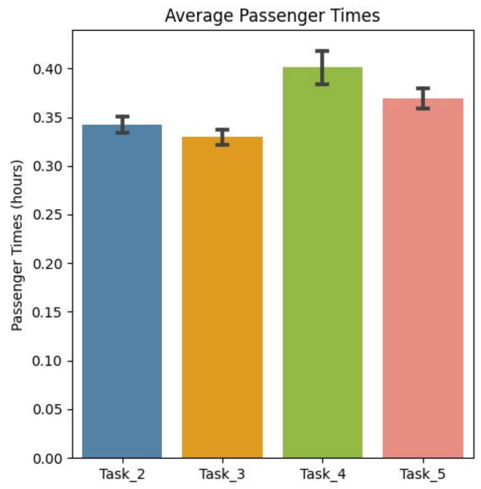
CS330 Case Study (NotUber)

Dmitri Morales, Jamie Wang, Karina Ng, & Cayla Park

Executive Summary

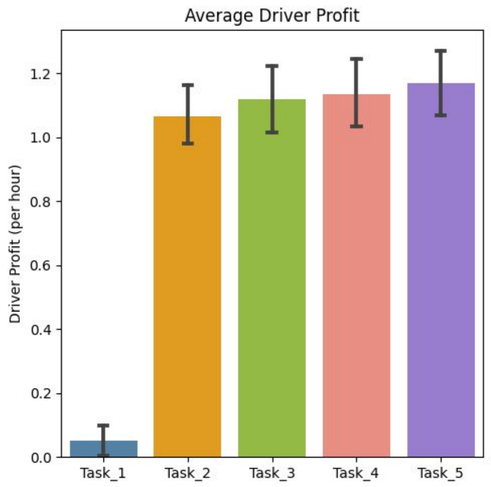
For this project, we were tasked with designing several implementations of **efficient automated algorithms** to match a list of passengers to a queue of potentially available drivers for our NotUber rideshare service. We were assigned several implementation tasks labeled T1-T5 with specific desiderata and constraints labeled D1-D3 and C1-C3, respectively. Refer to the case study document for specific details.



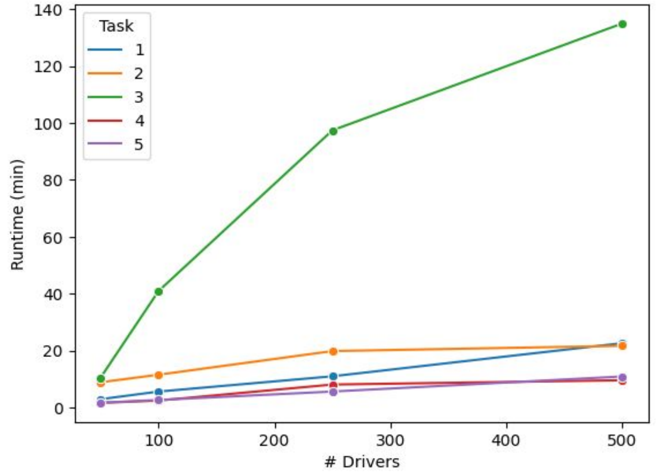
*note we excluded the average passenger wait time for T1 in this visualization because it is an extreme outlier.

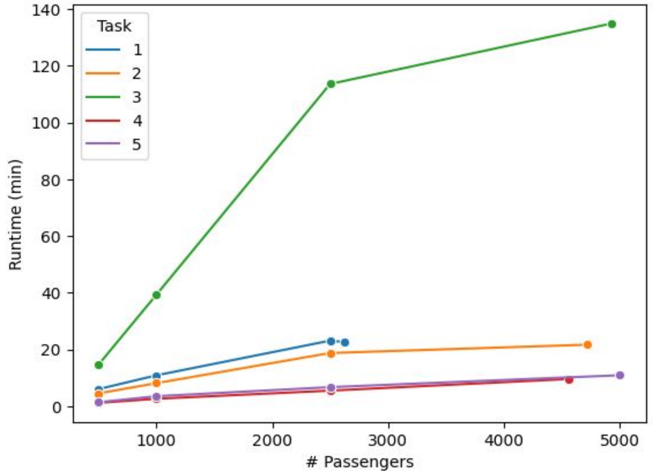
The results of the analysis we have conducted find that our algorithm for **T5** produced the best result in terms of optimizing the desired properties D1 (**minimized passenger wait time**), D2 (**maximized driver profits**), and D3 (**efficiency and scalability**).

On this slide, there are several visuals that highlight general trends of the findings of our case study. We will describe these outcomes in more detail in the slides to follow, but notice the upward **trend** in **driver profit,** and overview of average **passenger wait time**.



Executive Summary (cont.)





The graphs above illustrate the trends in the **runtime** of the number of drivers and number of passengers throughout each of the tasks. In the following slides, we will break down the analysis of the run times between tasks. Notice the drivers graph looks nearly linear, and the passengers graph resembles a slightly logarithmic trend. (However, these fits are not exact). For sake of runtime analysis, we arbitrarily chose to test at **10%, 20%, 50%, and 100% of all potential drivers and all passengers (and vice versa)** given to us in the *drivers.csv* and *passengers.csv* files.

T1

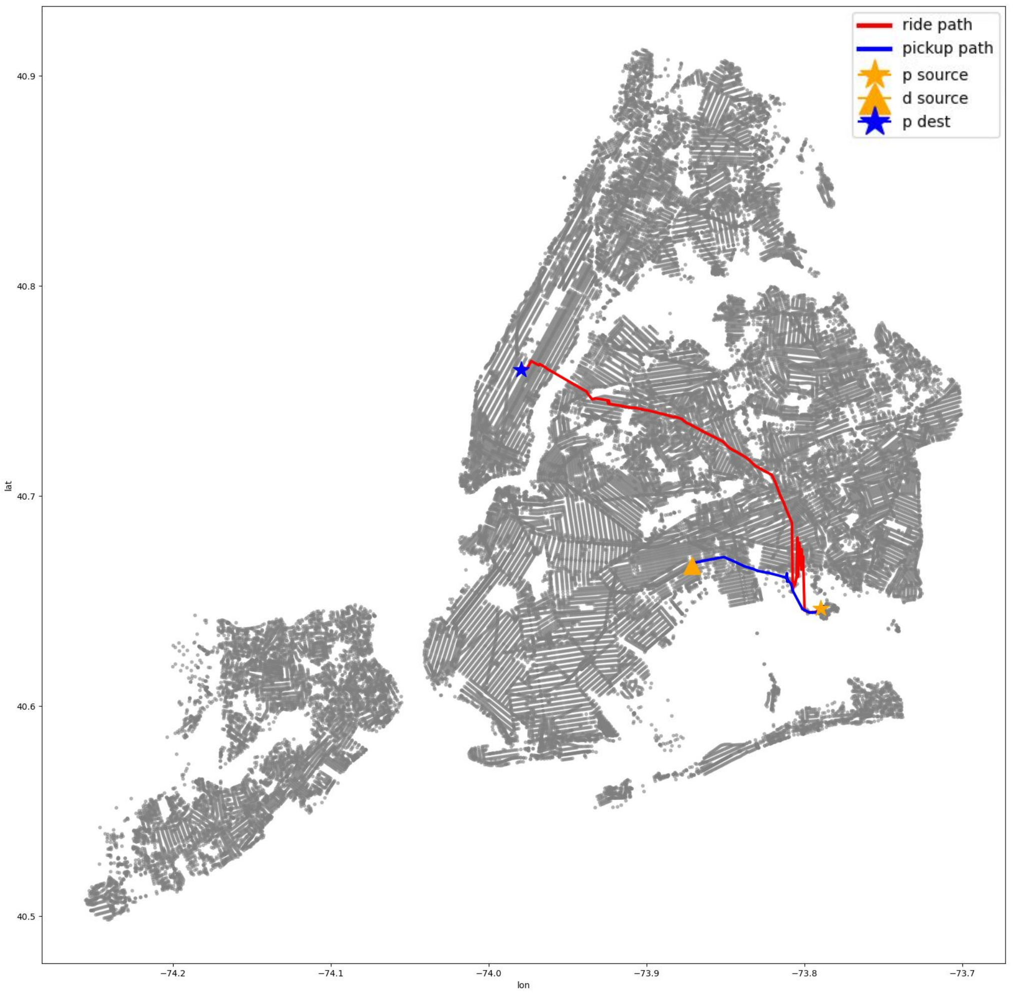
First Available Driver

Algorithm & Runtime Complexity

- Initialized the **list of passengers** and **queue of available drivers** by reading through the CSV file chronologically (since the passengers were already in order).
    - **Passengers**: This will take O(M* N) time where N is the total number of passengers, we loop through each passenger in the file (n=5000), and where M is the total number of nodes as we iterate through all nodes in the *node.json* file to find the nearest node.
    - **Drivers**: O(M*D) + O(M), where D is the total number of drivers, similar to above reasons, but we transform it into a heapq which is linear cost.
- **Updated the queue** of available drivers, heappop and heappush methods constant cost of O(1)
    - A driver is added to the queue if their available start time precedes the time a passenger enters the queue, a driver is added back to the queue upon completion of a ride
    - The **probability a driver will leave the queue** is determined by a randomly generated variable, and this distribution converges to 80% of the time the driver will stay in the queue
- For each passenger in the list, we match the passenger with the **first available driver in the queue**
    - The current time is updated to the next passenger's appearance time after each passenger is matched with a driver. After this, we applied Dijkstra's algorithm (explained on the next slide) to efficiently navigate the driver to the passenger's pickup location with the shorted path (with respect to time)
- If there are no drivers in the queue, the passenger must wait until a driver becomes available
    - This idle time is added to the passengers total wait time

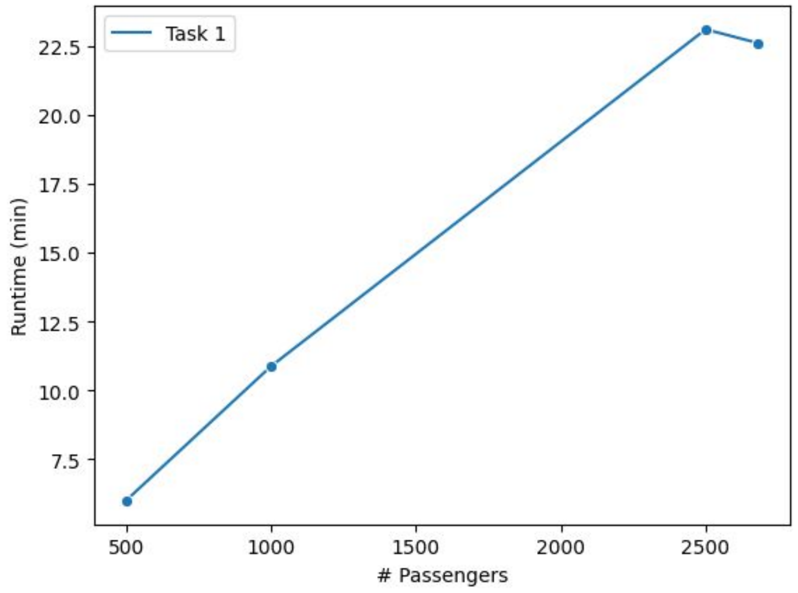
Visual Map of Dijkstra's Shortest Path

This is a visualization depicting the shortest path found by our implementation of Dijkstra's algorithm on the graph of nodes from the *node_ data.json* file. The **runtime** of Dijkstra's is expected to be **O((V+E)logV)** where V is the number of nodes on our graph and E is the total number of edges but with the density of our graph, we most likely experience the **worst case scenario runtime** of this algorithm which is **O(V^2logV).** As indicated by the key, the **yellow triangle** is the driver's starting location, the **blue path** is the driver's path to the pickup location, the **yellow star** is the passenger's pickup location, the **blue star** is the passenger's destination, and the **red path** is the route of the ride.



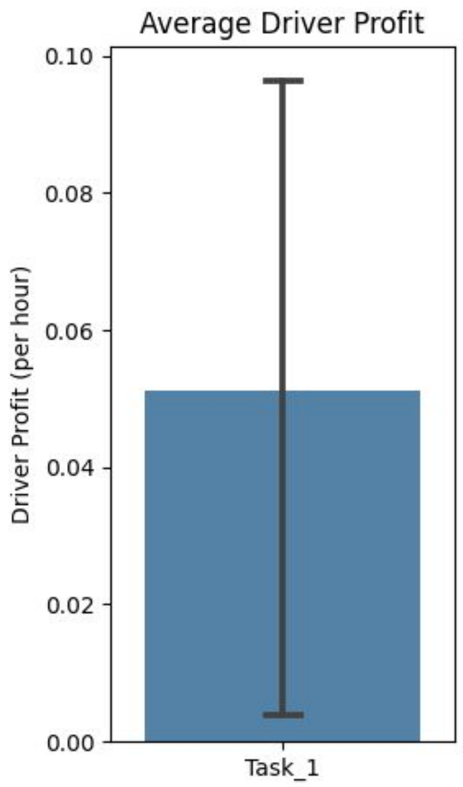
Runtime

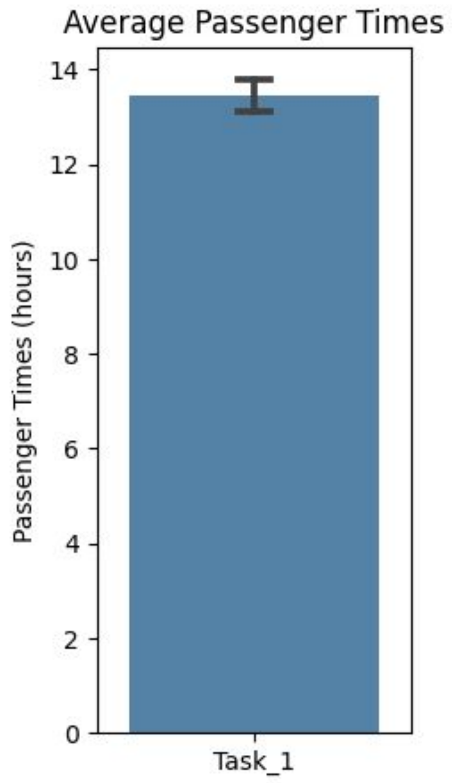
From our experimental tests, we can see altering the number of **passengers** and **drivers** does **scale with respect to the runtime**, however when we use 2500 passengers with the T1 algorithm, the runtime is slightly higher. This is because there are only 2628 passengers handled when using all passengers, so the runtimes should be approximately the same. The slight increase in runtime could be attributed to the *random.random()* returning a value that kicks different drivers out at different times.



Modeling & Experimental Results

- On average, the drivers earn a **profit of 0.05 hours** which is a little better than breaking even. Additionally, the average passenger time, which includes the time it takes to assign a driver from the time the passenger enters the queue to the time it takes the driver to reach the passenger is more than **13 hours**.
- This is clearly not ideal, but due to the nature of a driver potentially being on the other side of the map (NYC) and being matched with a passenger on the other end of the graph, traversal times will be long and unoptimized.



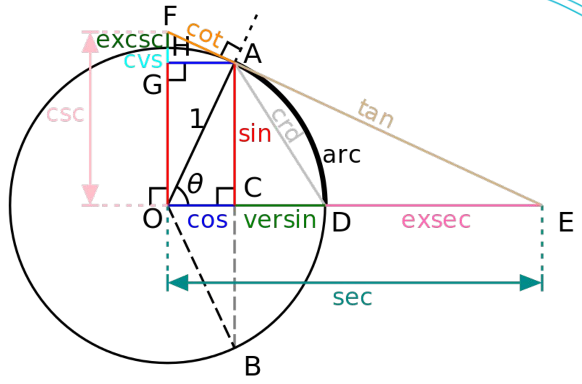


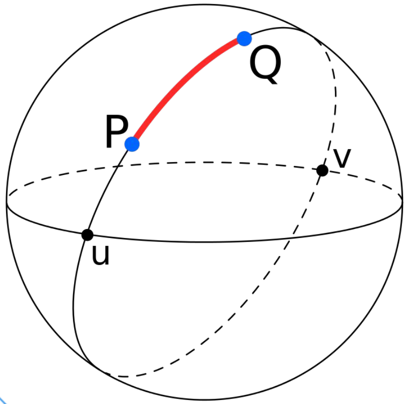
T2

Straight-line Distance

Haversine Formula

Rather than the straight-line planar Euclidean distance, the Haversine formula allows for **computation of distance along a sphere**, specifically we can apply it to find the distance between two specified points on the Earth's surface using their **latitude** and **longitude**.





The trigonometric function can be expressed as: $haversine(\theta) = sin^2(\theta)$

The central angle is

$$(d/r) = haversine(\Phi_2 - \Phi_1) + cos(\Phi_1)cos(\Phi_2)haversine(\lambda_2 - \lambda_1)$$

Where the radius (r) is of the Earth at 6371 km, and the distance (d) between two points Φ_1 and Φ_2.

Solving for the **distance**, we get

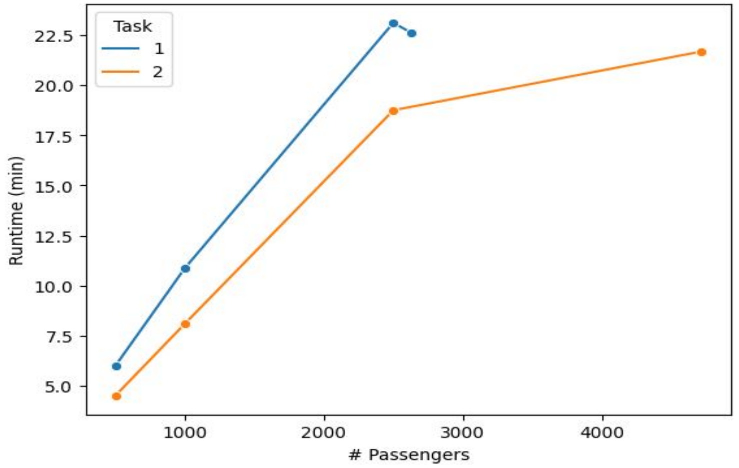
$$d = 2rsin^{-1}(\sqrt{(}sin^2(\frac{\Phi_2 - \Phi_1}{2}) + cos(\Phi_1)cos(\Phi_2)sin^2(\frac{\lambda_2 - \lambda_1}{2}))$$

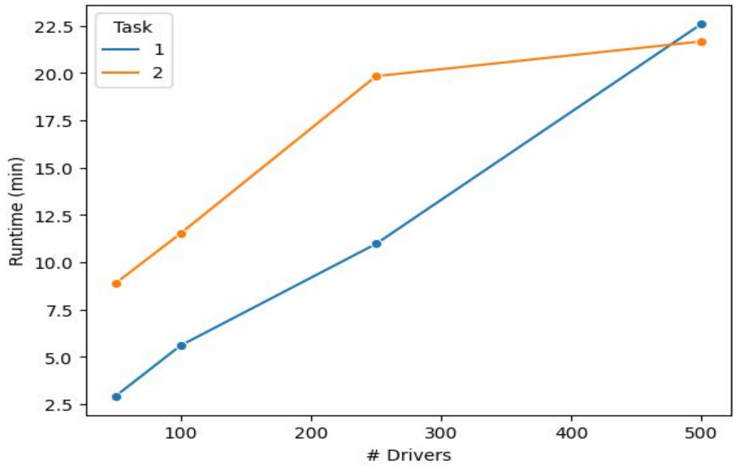
Algorithm & Runtime Complexity

- Initialized the **list of passengers** by reading through the CSV file chronologically (since the passengers were already in order). The runtime is the same as the initialization in T1.
- Implemented the same iterative process for the **driver queue** as T1
    - To improve upon our T1 algorithm, the probability a driver will exit the queue is dependent on the quotient of the hours worked by the driver divided by 8 hours (which we arbitrarily chose as an average work day). This dynamic approach still **randomizes the probability a driver exits** the queue.
    - Once the driver has been active for 8 hours, they are taken out of the queue of available drivers (they will complete their current ride and not be added to the queue if they are currently assigned a passenger)
- Utilized **Haversine distance** formula to compute the straight-line distance between two points given two points specified by latitude and longitude in the adjacency list, this computation is constant O(1)
    - This was applied to compute the distance between the passenger pickup position and destination node
    - Also computed the distance of the driver's current position to the passenger's designated pickup location
- Passengers are matched with drivers in the available queue based off a minimization of the straight-line distances from the driver's current location and the passenger's desired pickup location, this takes O(n) time if n is the number of drivers currently available in the queue.

Runtime

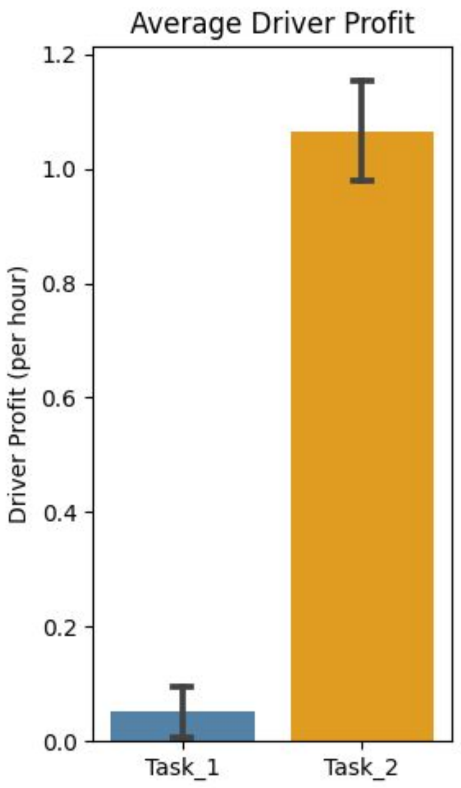
In these sets of tests, by altering the number of passengers in the dataset, we can see that **T2** on average is able to finish **running faster than T1**. Not only does T2 have a faster run time, it also **matches more passengers** when using all 5000 passengers. T2 matches 4718 passengers, whereas T1 only matched 2628. Matching passengers with drivers based off of **straight-line distance** shows to match the passengers **more effectively** than just matching passengers based on first available driver.

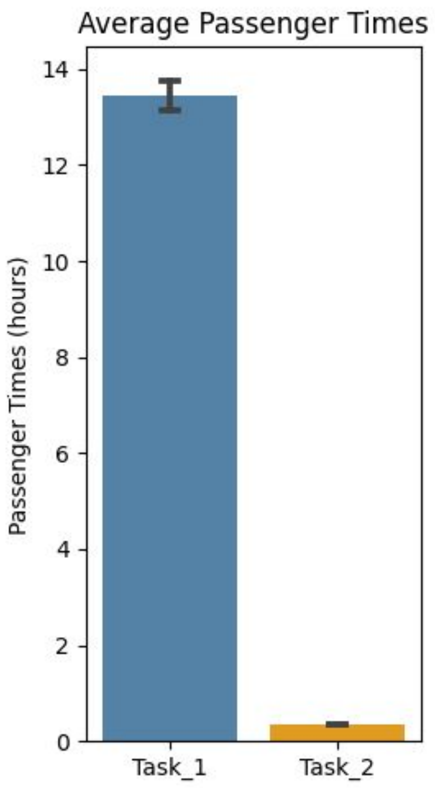




Modeling & Experimental Results

- Just by altering our implementation to match drivers with passengers by using the Haversine distance formula, we see a large increase in driver profit to more than **20x the profit** (~0.0511) from T1 to around **1.066**
- Additionally, the average **passenger time significantly decreases**, such that our passengers aren't waiting half a day for their ride (around 13.45 hours), but instead, a modest **21 minutes** approximately





T3

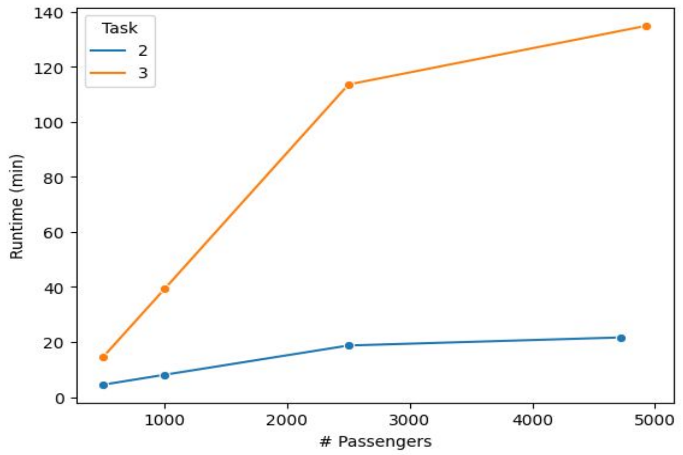
Minimum Estimated Time

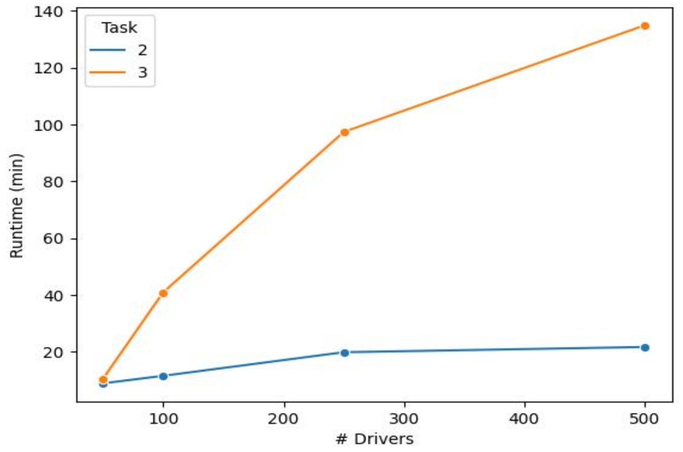
Algorithm & Runtime Complexity

- Initialized the **list of passengers** by reading through the CSV file chronologically (since the passengers were already in order). The runtime is the same as the initialization in T1.
- Implemented the same iterative process for the **driver queue** as T1, with same probability logic of the driver exiting the queue as T2, and forced exit after an 8 hour shift
- Utilized **Dijkstra's shortest path** algorithm from the driver's current location and passenger's designated pickup location.
    - The time (length/max speed) provided in the adjacency list is used as the weights of the edges
    - Finding the shortest path of the graph with time as the edge weights inherently minimizes the time to find a driver-passenger pair by nature of Dijkstra's algorithm
    - The runtime algorithm for Dijkstra's algorithm is **O((V+E) logV)**; however, because of a road-network like NYC being very dense, the runtime is probably closer to **O((V^2)*logV)** where V is the number of vertices/nodes in our graph and E is the number of total edges/roads in our graph/map.
- Passengers are matched with drivers in the available queue based off the shortest path, or a **minimization of** the **estimated time** difference from a driver's current location to the passenger's desired pickup location

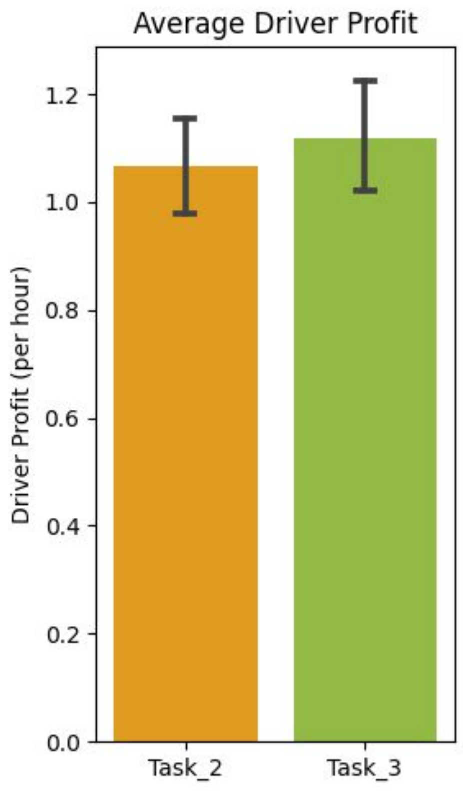
Runtime

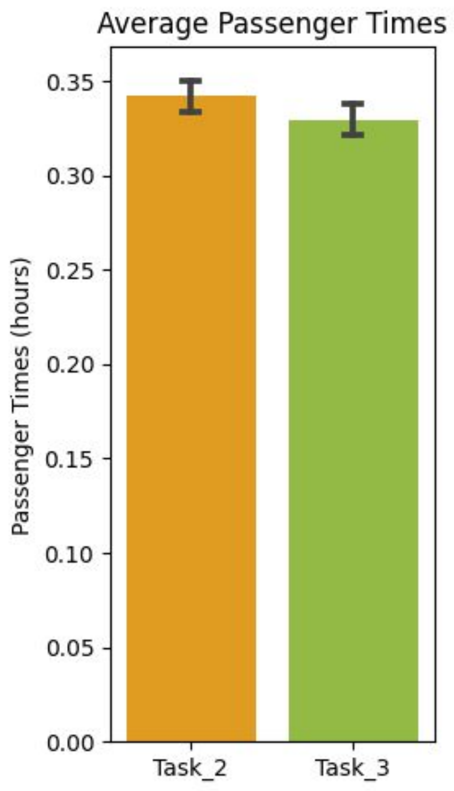
- Due to the dense structure of the adjacency list and nodes for mapping NYC, it is likely we experience the **worst case runtime** from Dijkstra's. This worst case runtime, though offering a better driver profit and passenger wait time isn't ideal for an app to match passengers with drivers.
- We could have improved this by implementing a data structure such as a dictionary with boolean values to keep a **record of visited nodes**, thus if we have already run Dijkstra's on a visited node we could just parse through this structure to find that value from prior iterations rather than recomputing Dijkstra's.
- Due to the nature of our algorithm implementing a heap structure instead of a priority queue, we were unable to implement an optimized version of this algorithm in the allotted time frame of this project. In future iterations to improve upon the runtime of our algorithm, this is definitely something to be considered.





Modeling & Experimental Results





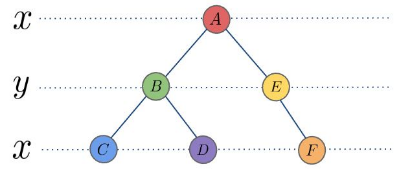
- In this scenario, by using **Dijkstra's algorithm** to match the available drivers and passengers instead of straight line distances, we see an increase in the average driver profit and a decrease in passenger wait time.
- From the straight-line distances in T2 to Dijkstra's shortest path in T3, there is approximately a **0.052 increase** in driver **profits** and approximately **one minute decrease** in average passenger **wait time**. Due to the density of the nodes in our graph, this marginal increase is reasonable.

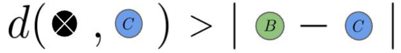
T4

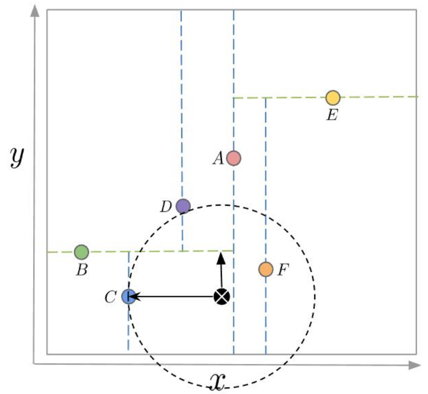
Optimized Runtime

K-Dimensional Tree and Spatial Indexing

A K-D Tree is a data-structure used to organize points in a K-dimensional space, where K is typically a very large number. In this case, we can use a K-D Tree to **efficiently find the nearest driver** to a passenger or vice versa. We are using 2 dimensions, **longitude** and **latitude**, to construct the tree.





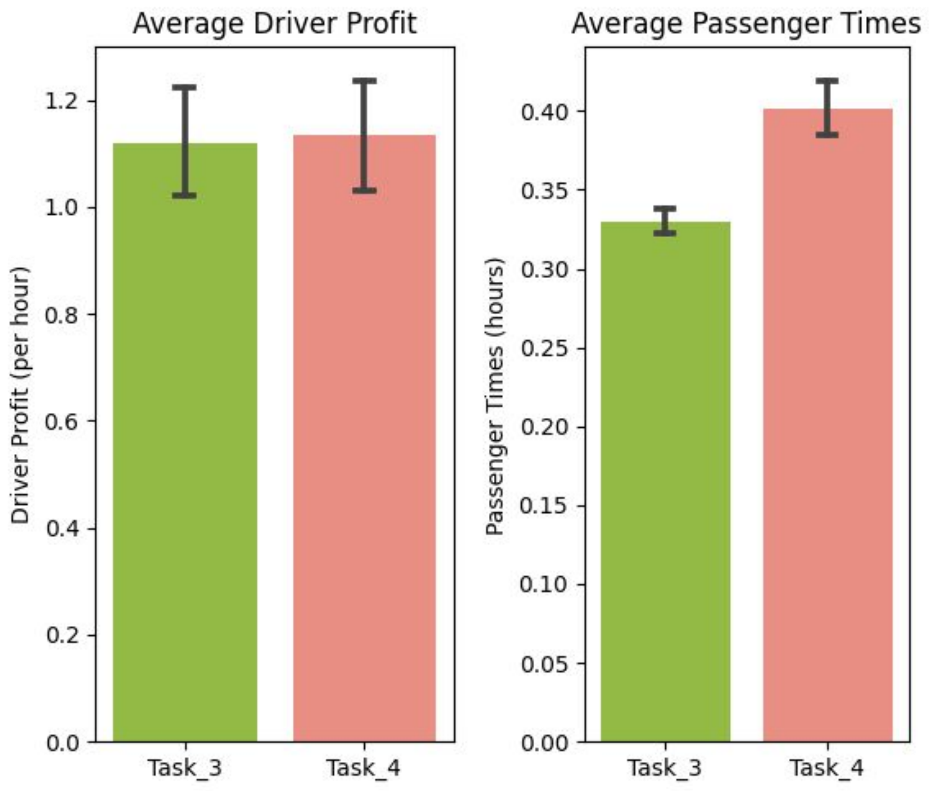


The structure of a K-D Tree is essentially a **binary tree** where each non-leaf node divides the space into two parts. It acts as a **hyperplane** that is perpendicular to some chosen axis, in this case longitude or latitude. To build the K-D tree, we recursively partition the points in the space, alternating between axes, to form the binary tree. In our creation of the passenger queue, if we say we have M passengers and N nodes for NYC, assuming a balanced tree, the creation of this queue would be O(M*log(N)). In worst-case scenario, with a highly unbalanced tree, queue creation would be O(M*N). Additionally, our implementation of the **KD-Tree was successful.** When we ran a small Python script to check the differences in passenger and driver node placement in comparison to the brute force method, we found no differences.

Algorithm & Runtime Complexity

- Initialized the **list of passengers and drivers** by reading through the CSV file chronologically (since the passengers were already in order). Used spatial indexing to initialize a **K-D Tree** to pre-process the nodes to efficiently find the nearest node for each driver and passenger. The runtime for finding the nearest node is O(logN) and worst case would be O(N) if the tree is unbalanced.
- Implemented the same iterative process for the **driver queue** as T1, with same probability logic of the driver exiting the queue as T2, and forced exit after an 8 hour shift
- Utilized **A*** algorithm from the driver's current location and passenger's designated pickup location
  - The **Haversine formula** was implemented as our **admissible heuristic** because the straight line distance will never overestimate the actual distance it takes to traverse the graph from start to destination node
  - In order to account for the inconsistency in units (time vs. distance), the result of the Haversine computation was divided by the **average road speed** for the current hour and current day which was determined by the speeds provided in the adjacency list– this new scaled value became our heuristic
  - The time (length/max speed) and heuristic value were used as edge weights to determine the shortest path
  - Finding the shortest path of the graph with time as the edge weights inherently minimizes the time to find a driver-passenger pair. Assuming we created an **admissible heuristic**, the runtime of A* is minimized to O(D) where D is the distance from the start node to end node. In worst case, it would be the same as Dijkstra's O((V+E) log(E)) where V is the number of vertices, and E is the number of edges.

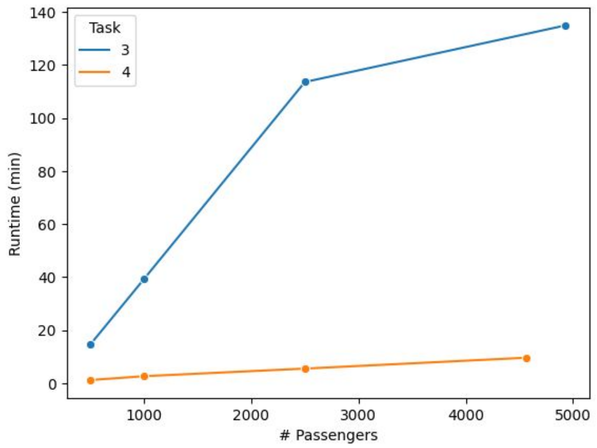
Modeling & Experimental Results

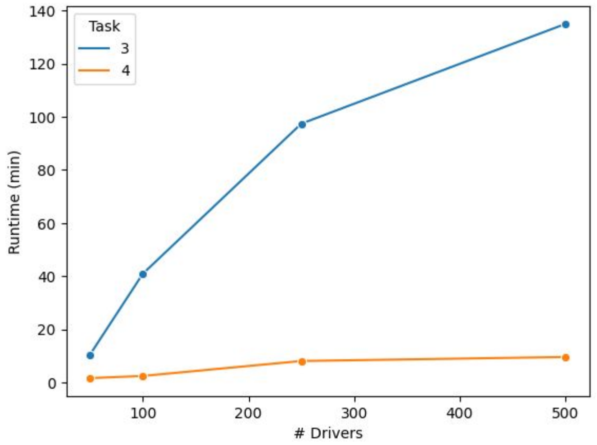


Though there was a slight increase in average driver profit, the average passenger time graph indicates a fault in our shortest path algorithm. Despite **factoring in different average speeds** based on the time of day and day type, our heuristic had slight fault leading to **4 extra minutes** of total **passenger wait and ride time**. In the future, we could potentially implement a dynamic heuristic that uses historical data to better optimize our A* algorithm to minimize Passenger Times.

Runtime

- The implementation of our **A*** algorithm significantly **reduced the runtime** across the board as we were leveraging the heuristic to dictate our shortest path. We also made it scalable and efficient with number of passengers and drivers increasing over time. Using the full passenger list, we were able to process 4600+ passengers at **9 minutes and 35 seconds** compared to the **2+ hour** runtime for T3.
- Preprocessing of the nodes using the K-D Tree also significantly **reduced runtime** of the initialization of the passenger list and driver queue. When using the brute force implementation, the **passenger list** took around **6 minutes** to match each passenger with the nearest node, whereas using the K-D Tree came in around **1 minute**. For the **driver queue**, instead of it being **35 seconds**, it went down to **2.6 seconds**.





T5

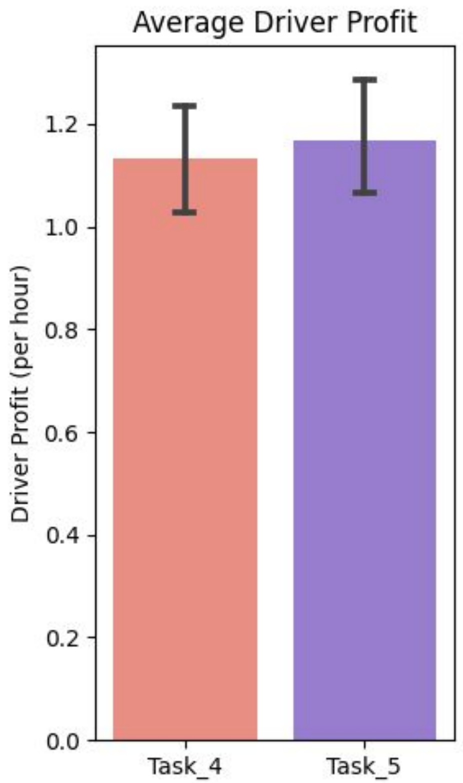
Improved Scheduling Algorithm

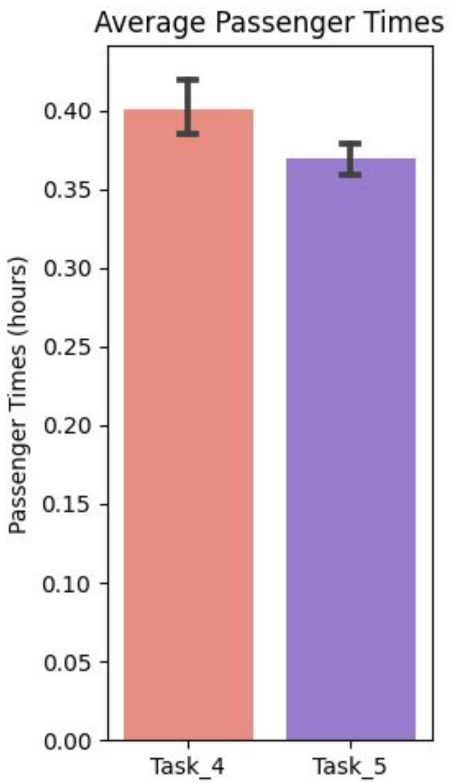
Algorithm & Runtime

- Initialized the **list of passengers and drivers** by reading through the CSV file chronologically (since the passengers were already in order). Used spatial indexing to initialize a **K-D Tree** to pre-process the nodes to efficiently find the nearest node for each driver and passenger
- Same probability logic of the driver exiting the queue as T2, and forced exit after an 8 hour shift
    - Attempted to use an exponential dynamic analysis of whether or not a driver would exit, but this led to an increased runtime since the queue would be bigger on average
- Utilized **A*** algorithm from the driver's current location and passenger's designated pickup location
- To enhance scheduling algorithm, an **dynamic extended search time** was implemented
    - The initial extended search time was defined as the minimum pickup time for an available driver (if one exists) to travel to a passenger
    - A better driver match is then searched for within this extended search time after the passenger request has been made
    - A better driver match exists if the wait time for a particular driver to become available combined with the time needed for that driver to pick up the passenger is less than the extended search time
        - This driver then becomes the best match and the extended search time becomes that combination of wait time and pickup time

Modeling & Experimental Results

- From our testing, we saw a very similar trend in runtime from T4 to T5. We achieved the full 5000 passengers, to which the second closest algorithm was T3 with 4928 passengers.
- Not only this, but our modifications in the T5 dynamic scheduling led to the **slight decrease in passenger wait time** from T4 and maintained driver profit.

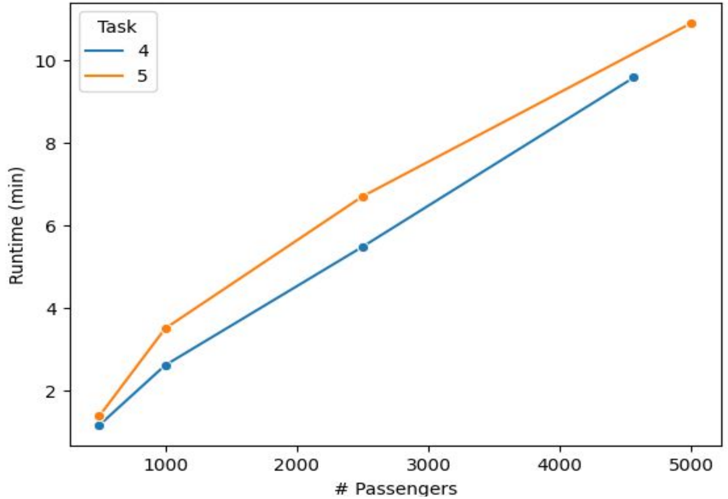


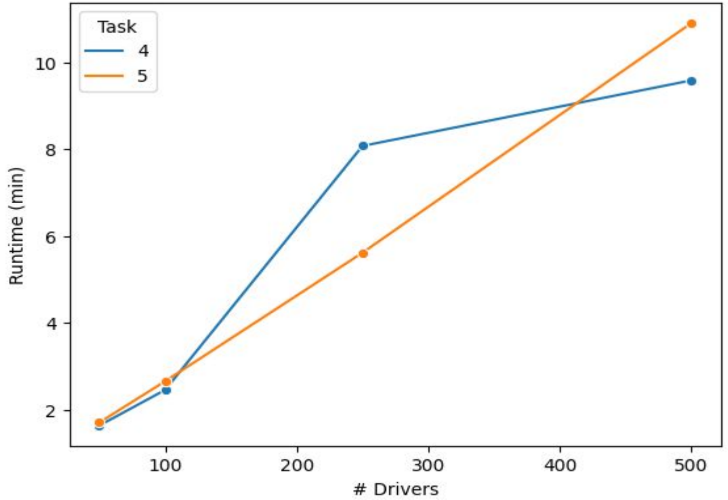


Runtime

After implementing the improved scheduling algorithm, we can see that the runtime for **T5** is on average **slightly slower than T4.**

This is due to the extended search space and therefore additional A*s performed to find the best driver.





B1-B4

Additional Bonus Questions

B1: Surge in Demand

Q: So far we have considered passenger performance by arrival time, perhaps the total or average. What about when there is a surge in demand, and there are many more passengers requesting rides than you can match to drivers? How could you design your algorithms to be fair to passengers and cope with such high demand?

A: One way that we can cope with high demand, is by selecting different factors to prioritize certain passengers over others rather than just arrival time. These factors can include indicating urgency of a ride or implementing a passenger rating system to prioritize certain customers who have habitually used the NotUber service in the past or have positive ratings from previous drivers and rides. In our current iteration of the algorithm, we already account for driver proximity and passenger wait time, but to cope with higher demand and remain fair, we can modify aspects of our matching algorithm such as rating or urgency to handle the surge. A combination of all of these factors could be calculated into some sort of priority score, which will be how we identify which passengers will be able to ride first.

We can also add a factor of dynamic pricing so that during times of high demand, the cost of rides for passengers will be higher, meaning that earnings for drivers will be higher. There can be a base cost that is calculated by the time and distance of the ride, which then may increase or decrease depending on demand and traffic. This will help control demand on the passenger side, but increase demand on the driver side. The higher prices would decentivize passengers to ride at the time and incentivize more drivers to drive because of higher earnings. Another possible aspect of this, can be the option for customers to be prioritized for a ride by paying a fee. This will allow customers that have an urgent ride and are willing to pay to be prioritized, and drivers would earn more as well. In calculating the total fee, we can create some factor that is based off of demand and multiply that by the base cost. The prioritized passengers can be kept in some boolean value that is True for when they are prioritized and False when they are not.

If there is too little supply of drivers, there can also be a ride-share option, where the passenger can indicate if they are open to sharing the ride with another passenger for some discount. These passengers can have the same final destination, or the driver could make multiple stops in this case, serving as a smaller scale private transportation service. In this case, we would have to consider many factors such as capacity of the vehicle, and ideal routing to destinations that also passes through pickup locations for each passenger.

B2: Prioritize Driver Ride Profit

Q: We have primarily emphasized passengers and runtime efficiency. Suppose you have a competitor NNU (Not-NotUber) that opens for business in the same city. NNU claims that its scheduling algorithm is better for drivers and many of your drivers leave to join NNU. NNU claims that its algorithm does a better job of optimizing D2, the driver ride profit, and they claim it is also more fair to drivers, ensuring that drivers get a more equitable number of rides assigned to them. How could you modify your algorithm to compete with NNU for drivers while still maintaining good performance for passengers?

A: In order to compete with NNU for drivers, we will have to prioritize driver ride profit more. In order to ensure that drivers get a more equitable number of rides assigned to them, we can keep track of their ride history and some average of their profits. This ride history would keep track of the average distances of each of their rides, which is essentially profit. This will allow us to make sure that drivers are getting a good distribution of short-rides that are less profitable, and long-rides that are more profitable.
We can also consider allowing drivers to have a preference of certain areas, which could minimize the amount of pickup time as they are restricted to an area and it would also increase driver satisfaction. By minimizing the area that they are assigned to, it also makes the matching process more efficient as we would be looking at a known set of drivers that are in that area for some passenger. For areas that have too little drivers but still have demand from passengers, we can identify them as hotspots, and increase the pricing for those areas to attract more drivers.

In order to implement this, we would need some database that can keep detailed information about each ride as well as their drivers, in order to calculate metrics such as average earnings per ride, the distribution of ride distances, and popular pickup/destination areas. Using this information, we can consider also giving drivers a priority score that is calculated based off of their average earnings per ride. To ensure that we fairly distribute drivers but also maintain good performance for passengers, proximity to the drivers would have a higher weight than other factors like average earnings per ride. This factor would come into play in situations where if there are drivers that are all around the same distance from the passenger, the driver with the higher priority score would be matched to the passenger.

B2: Prioritize Driver Ride Profit

Q: We have primarily emphasized passengers and runtime efficiency. Suppose you have a competitor NNU (Not-NotUber) that opens for business in the same city. NNU claims that its scheduling algorithm is better for drivers and many of your drivers leave to join NNU. NNU claims that its algorithm does a better job of optimizing D2, the driver ride profit, and they claim it is also more fair to drivers, ensuring that drivers get a more equitable number of rides assigned to them. How could you modify your algorithm to compete with NNU for drivers while still maintaining good performance for passengers?

A: Another consideration is potentially having the passenger 'walk' to a nearby node, and check to see if a driver can traverse faster to said source location. This is in the event of certain edges containing lots of traffic, and if these edges can be avoided, this would not only increase driver profit, but keep passengers satisfied. We would keep the walking distance within a reasonable range, where the node the passenger walks to would be closer to the driver, and to a node whose path to the destination would avoid more traffic. We can use the same K-D Tree we currently have implemented to identify the closest nodes to the passenger source. We would then need to identify the estimated driving time from these nodes to the destination based off of real-time traffic data to choose the node with the least exposure to traffic. This would in turn reduce the driver's pickup time as well as minimize encounter of traffic.

B3: Prevent Congestion and Road Traffic

Q: Suppose your service becomes so popular that half of all road traffic comes from your drivers. You notice an issue where sometimes your algorithm assigns too many drivers at once to the same routes causing congestion and traffic stops on the road network. How could you modify and expand your algorithm to account for congestion caused by your own drivers?

A: To modify and expand our algorithm to account for congestion, we can account for real-time traffic conditions in our routing algorithm. The route does not have to be static, and can instead be updated during the ride as well based off of traffic conditions. This can be integrated into our current algorithm by implementing a load-balancing mechanism, which would distribute the drivers more evenly across different routes. One way to implement this is by assigning weights to certain routes that represent their load or congestion. Then, when a new driver becomes available, we iterate through the available routes while considering their congestion weight. We would then select the route with the lowest load, or a load below our defined threshold, and assign the driver to that route. To account for real-time changes in traffic, we would want to dynamically adjust these weights by regularly updating them with the current traffic data (how many drivers are scheduled to be on that route at a given moment in time). This way we can monitor if a route is becoming too congested and evenly distribute the drivers along all given paths. We would still be maximizing profit with this approach because if a driver is able to complete their ride by taking a slightly longer path but finishing in significantly less time, the driver would then be able to complete more rides and generate more profit for the duration of their shift.

Another alternative would be to implement adaptive scheduling and stagger departure times of drivers so we can minimize the influx of drivers on any specific route. The downsides to this modification is that the idle time of drivers would potentially increase, and by consequence the waiting time of passengers would increase as well. However, this could minimize the actual time a passenger is in the car (with the reduced traffic congestion), and as long as the extra time a passenger has to weight is less than the additional time it would take the passenger to reach its destination with the traffic congestion, this could be a better optimization.

B4: Incorporate Historical Data

Q: After your service has been running for several years, your data science team decides to optimize your scheduling and routing algorithms based on years of historical data to make more informed decisions. How will you incorporate historical data into your algorithm?

A: The historical data would be very helpful on determining the shortest path to a destination. When we have historical data implemented, we'd be able to look at trends in different hours of the day at different times of the year to anticipate what the edge weights would be, to potentially traverse through a different road to get to a destination. This would be similar in implementation to the algorithms described in B3. Additionally, it may assist drivers in knowing where many people would be requesting rides from, so that drivers can linger in this area and minimize drive time to passenger source node. In this case, we would allocate a designated amount of drivers to certain areas of the map, and restrict the search algorithm to the the k-nearest nodes in proximity to the driver's current location with the passenger's destination also being within reasonable distance from this area. This way the driver will not stray far from their starting location and can complete rides in a more efficient manner. An example of this would be our NotUber app knowing that during the Thanksgiving season, many students from Duke will be requesting rides to the airport. Our app would be able to notify drivers to stay around the Duke area as there is a high likelihood of passengers requesting rides to the airport. Additionally, if certain roads are particularly busy during the Thanksgiving season, drivers can be notified that there exists a shorter path with less assumed traffic on the path to the airport.

Another thing that historical data could assist with is creating dynamic pricing for rides. To implement this, we would just scale up the profit of a driver's ride to incentivize more drivers to join the app/queue when there is an expected surge in passengers. For example during the said Thanksgiving break, since we'd know that there would be many individuals requesting rides from Duke's campus, we could purposefully increase the ride price to maximize profit for the driver, because of the increase in demand from a certain area.