CS330: Case Study Report

Kushagra Ghosh, Matthew Rui, Haiyan Wang, Felix Zhu

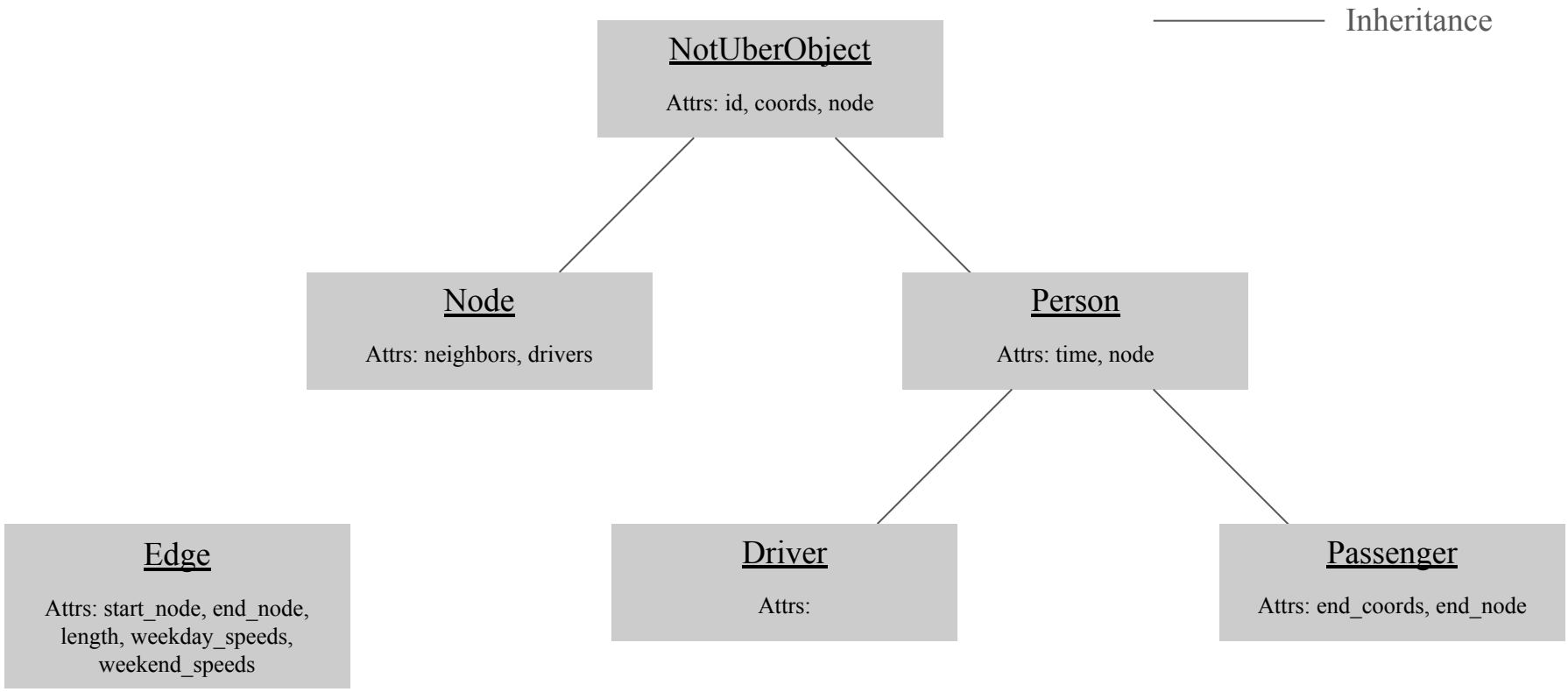
Executive Summary

We design and implement several classes to model this ride service on the road network in New York City. We implement a matching algorithm that assigns drivers to passengers and simulates their rides through the city, and at each step in this case study, we introduce further complexity to the algorithm to improve its simulation accuracy while maintaining scalability and efficiency. In particular, we improve the algorithm in three main areas:

- Efficiency of locating drivers and passengers in relation to the road network (i.e. assigning drivers and passengers to nearest nodes/possible pickup locations)
- Finding the nearest driver to a given passenger
- Finding the shortest path between two points on the network (for pickups and dropoffs)

We find that there exists a tradeoff between accuracy of simulation and runtime efficiency. However, with the proper space partitioning and graph traversal methods, we can maintain a reasonably scalable simulation with a high degree of accuracy.

Class Structure



Class Structure - Important Methods

NotUberObject

- __eq__(self, other) - Compare object type and id to determine equality
- __hash__(self, other) - Return object id as hash code (sufficient for our purposes)
- euclidean_dist(self, other) - Return euclidean distance between lat/lon coordinates

Node

- shortest_path(self, end_node, start_time) - Dijkstra's algorithm on network at given time
- shortest_path_a_star(self, end_node, start_time, avg_mph) - A* on network at given time with estimated Euclidean travel time (Euclidean distance / avg_mph) heuristic
    - avg_mph is the average speed limit across the entire network
- partition(self, grid, grid_params) - Partition nodes into grid of subpartitions with preset parameters for faster searching and driver/passenger assignment

Person

- partition(self, coords, grid_params) - Find subpartition in which Person (passenger or driver) appears
- assign_node(self, coords, grid, grid_params) - Find nearest node to Person (given that passengers and drivers aren't guaranteed to appear at nodes on the network)

Initialization

Nodes - Generate Node instances from parameters given in node_data.json

- Stored in dictionary NODES: <node_id, node_object>
- T4/T5: Define rectangular area based on edge nodes (least/greatest lat/lon coordinates in network), partition nodes into prespecified number of rectangular subpartitions (all with the same area)

Edges - Generate Edge instances from parameters given in edges.csv

- Store Edge objects in Node.neighbors attribute (start_node of edge_object)
- As edges are initialized, calculate average speed limit across all edges and times

Drivers/Passengers - Generate instances from parameters given in drivers.csv and passengers.csv

- Stored in lists DRIVERS and PASSENGERS (later converted to priority queues by time)
- T3: Search all nodes and assign each person to nearest node
- T4/T5: Assign each person to a subpartition in the previously defined grid, search only the nodes in the 3x3 section around that driver's subpartition to get nearest node
- Note that initializing all drivers/passengers at once and storing them in priority queues by appearance time is the same as simulating the passage of time and initializing them as their requests are made

Metrics

D1: Passenger Wait Time - the amount of time between requesting a ride and arriving at the destination. This includes time for drivers to appear, time for drivers to drive to the passenger, and time for drivers to drive to the destination.

D2: Driver Profit - the amount of time drivers spend driving with a passenger minus the time spend driving without a passenger.

D3: Robustness and Scalability - not a quantifiable metric, but we analyze each algorithm in light of its ability to accurately model and to efficiently simulate.

Modeling Assumptions and Limitations

Several assumptions and concessions must be made on the data in order for us to reach a (feasibly) simulatable environment.

- Driving times are estimates. Since we are only provided average speeds, we are inherently inaccurate about driving times. Additionally, when calculating our paths on the graph, if our journey takes us between hours or days, the average speed may change and not match our estimate exactly.
- Drivers will leave, at some point. We use a geometric random variable with p=1/15 after each ride completes, where a "success" is the case where a driver stops driving. This means we expect each driver to perform 15 rides before signing off.
- Drivers and Passengers "snap" to nodes. Just like in real ride services, only certain locations can be used to pickup and drop-off passengers, so we assume a driver or passenger will go to these locations and disregard the time taken to reach the nodes.

Notation: Let D be the number of drivers, P be the number of passengers, and N be the number of nodes. Note that in some cases, when referring to the road network graph, we will also use |V| and |E| to denote the number of nodes and edges, respectively

T1

We match each passenger (in chronological order of ride request time) with the first available driver

- If no driver is available, we measure the amount of time passenger waits before a driver becomes available
- If driver was available before the request, we measure the amount of time driver waits before being assigned
- We estimate drive times by calculating Manhattan distance between start and end points and dividing by the previously calculated average speed limit (a very approximate metric)
- When the passenger has been dropped off, the driver coordinates are updated to passenger's destination, time is updated to reflect driving time, and driver is reinserted into the available drivers queue with probability 14/15 (we expect a driver to give 15 rides/night)

Average Passenger Wait Time	Total Driver Profit	Average Driver Profit	Pre-Computation Runtime	Simulation Runtime
38.082 minutes	788.187 minutes	1.580 minutes	5.595 seconds	3.344 seconds

T1 Metrics

We expect that T1 passenger wait times are longer than future simulations. Because we match drivers only by availability (i.e. the driver that has waited longest is immediately matched) without considering distance to passenger, it is unlikely that we select the closest possible driver to each passenger when matching, and the passenger will need to wait longer than they do in future simulations.

T1 driver profit is harder to estimate without empirical data because it depends additionally on the time it takes for the driver to drop the passenger off. We estimate this value by calculating the time taken to traverse the Manhattan distance from start to end at the average speed limit across all edges, but for obvious reasons (not accounting for traffic, not exactly following the network), this is a very rough estimate and it is hard to say whether it is an underestimate or an overestimate (which depends on traffic variability and road geometry).

T1 is reasonably scalable. It runs in linear time, $O(P)$, with respect to the number of rides requested (assuming there are still available drivers). However, it remains a rough estimate and is likely not extremely accurate.

T2

We track all available drivers when a passenger requests a ride and match the driver closest to the passenger (by Euclidean distance)

- If no driver is available, we only consider the first driver to become available
- Other procedures are the same as T1

Average Passenger Wait Time	Total Driver Profit	Average Driver Profit	Pre-Computation Runtime	Simulation Runtime
27.653 minutes	-5765.001 minutes	-11.553 minutes	4.983 seconds	5.786 seconds

T2 Metrics

We expect that T2 passenger wait times will be shorter than T1. Because we now match drivers with both availability and an estimated metric of distance to passenger (Euclidean), we are far more likely to select a closer driver, if not the closest, compared to a matching algorithm that only depends on availability. Again however, an estimate based on Euclidean distance is likely to contain error from several sources mentioned previously.

T2 driver profit is again harder to estimate for the same reasons as T1. We are still estimating this value by calculating the time taken to traverse the Manhattan distance from start to end at the average speed limit across all edges.

T2 is also reasonably scalable. In the worst case, all drivers are available at once, and we iterate through all drivers for each passenger for an algorithm of $O(PD)$. However, in the real world including the dataset provided, this is an unlikely scenario, and we instead iterate through at most some upper bound c of available drivers for $O(cP)$. T2 also remains a rough estimate and is likely not extremely accurate. As we will demonstrate in later algorithms, there are far better methods of estimation and simulation that maintain a similar level of scalability.

T3

We use the same procedure as T2, but rather than measuring driver distance by Euclidean distance, run Dijkstra's algorithm to find the driver closest to a given passenger along the network.

- Theoretical Time Complexity: Since we used a binary heap implementation (Python heapq), the time complexity of our Dijkstra's algorithm is $O((|V|+|E|) \log(|V|))$.

Additionally, T3 introduces Dijkstra's to calculate the time taken to drive to locations, as opposed to the Manhattan distance estimates used in T1 and T2.

Average Passenger Wait Time	Total Driver Profit	Average Driver Profit	Pre-Computation Runtime*	Simulation Runtime
13.234 minutes	23882.852 minutes	47.766 minutes	2.621 seconds	6.5 hours (approx.)

*Pre-computation takes 121.742 seconds with node assignment, but we consider that to be part of the simulation

T3 - Modeling and Metrics

A significant difference in T3 compared to T1-2 is the actual usage of the graph in calculations. Whereas T1 and T2 only make guesses at the time it takes for drivers to go to and from places (constant time), T3 actually calculates the distances with Dijkstra's algorithm (though we still make certain approximations, such as assuming traffic is constant after a driver begins driving). Thus, we would expect T3 to offer a much more accurate simulation, though still with potential for error. We will treat our results in T3 as a baseline going forward.

Because we use the graph, we also notice a major increase in runtime. Whereas before, iterating through passengers was akin to running a single for loop, T3 introduces algorithmic complexity that adds overhead storage and computation which drastically slows our algorithm. Introducing Dijkstra's to estimate the driving times using provided speeds gives us a time complexity scaling on the number of drivers, nodes, and edges. Our time complexity gets kicked from $O(P)$ to $O(PD(|V+E|) \log|V|)$.

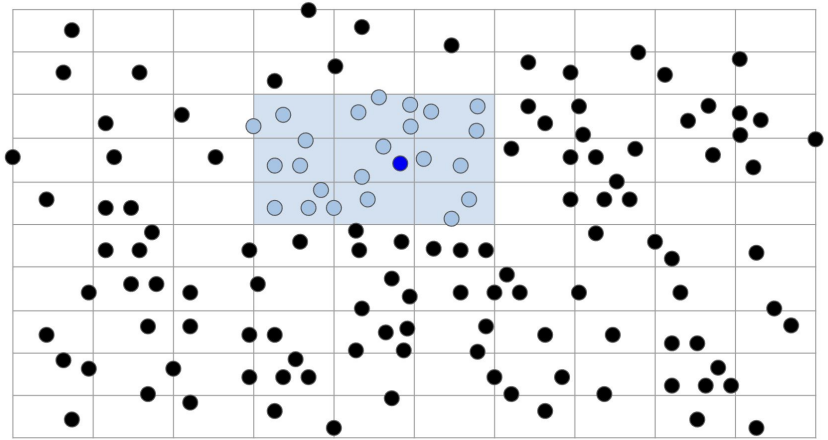
Based on empirical observation, this is NOT scalable. Although we model the situation better, our algorithm is painfully slow. As the numbers of drivers increases, the time for even a single iteration slows to a crawl, which in practice would be a terrible product (at higher scale, you may need to wait hours just for a driver match!). Also important to note is the node assignment algorithm. To assign drivers and passengers to nearest nodes, we iterate through all drivers and passengers and all nodes for $O(DN)$ and $O(PN)$ time complexity, respectively, which is highly inefficient.

T4

We use the same procedure as T3, but we improve the matching algorithm's performance by replacing Dijkstra's algorithm with A* in all instances. Our implementation of A* implements a heuristic that combines Euclidean distance and the previously calculated average speed limit across the network to calculate an approximate travel time to destination. In initialization, we also assign nearest nodes with *grid-partitioning* rather than by brute force.

Average Passenger Wait Time	Total Driver Profit	Average Driver Profit	Pre-Computation Runtime	Simulation Runtime
13.816 minutes	24576.948 minutes	49.154 minutes	17.483 seconds	1.5 hours (approx.)

T4 Partitioning



Assume we're given the network above partitioned into a 10x10 grid where black points represent nodes and the dark blue point represents a person (who does not appear at a node). To find the nearest node to the person, we only need to search nodes in the surrounding 3x3 grid of subpartitions (light blue), rather than looping through all nodes. On average (or assuming the network is distributed reasonably uniformly), this drastically improves assignment time. If the surrounding doesn't contain a node, we expand radially (i.e. 5x5, 7x7, …) until we find a node.

Let M be the total number of subpartitions. If the nodes are perfectly uniformly distributed, we expect to find $9N/M$ nodes in the 3x3 grid, scaling the node assignment procedure's runtime complexity down by a constant factor. In testing, we observed runtimes decrease by a factor of 5-6 on all our machines.

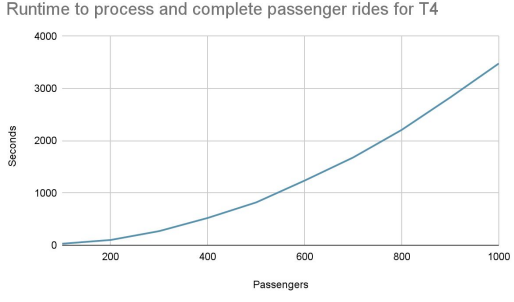
Theoretical Time Complexity of T4 vs T3

The worst case time complexity of A* (used in T4), is $O(b^d)$, where b is the branching factor and d is the length of our shortest path between our source and destination locations.

- However, in our case study results, A* performed much better than the worst-case bound since we had a heuristic function that was admissible and consistent based on euclidean distance ÷ average speed, we implemented early exit, and we used a queue as part of our implementation for efficient removal and insertion.
- If our heuristic function assigned a uniform cost heuristic (for example assigning the value of 0), then the time complexity of A* (T4) would be the same as Dijkstra's algorithm (T3), but since we used a better heuristic function, the runtime for T4 using A* is quicker in practice than T3.

For our overall *runtime* for T4, A* is invoked each time we calculate the time taken for a driver to drop off a passenger, and also each time we calculate the time taken for the closest driver to arrive at the passenger's location before a ride.

- As the number of available drivers increase in further timepoints of our simulation, our runtime to handle each passenger ride also increases, since the greater number of available drivers means more function calls to A* to compute the closest driver and drop off times, increasing our runtime of T4.




```
Finished initialization, total time 24.819733381271362 seconds
Time for 100 passengers: 28.501189708709717 seconds
Time for 100 passengers: 71.7676272392273 seconds
Time for 100 passengers: 169.38103485107422 seconds
Time for 100 passengers: 251.9543797969818 seconds
Time for 100 passengers: 297.98053336143494 seconds
Time for 100 passengers: 418.2118196487427 seconds
Time for 100 passengers: 442.26002526283264 seconds
Time for 100 passengers: 528.0879819393158 seconds
Time for 100 passengers: 620.8492932319641 seconds
Time for 100 passengers: 648.8344168663025 seconds
```

T5

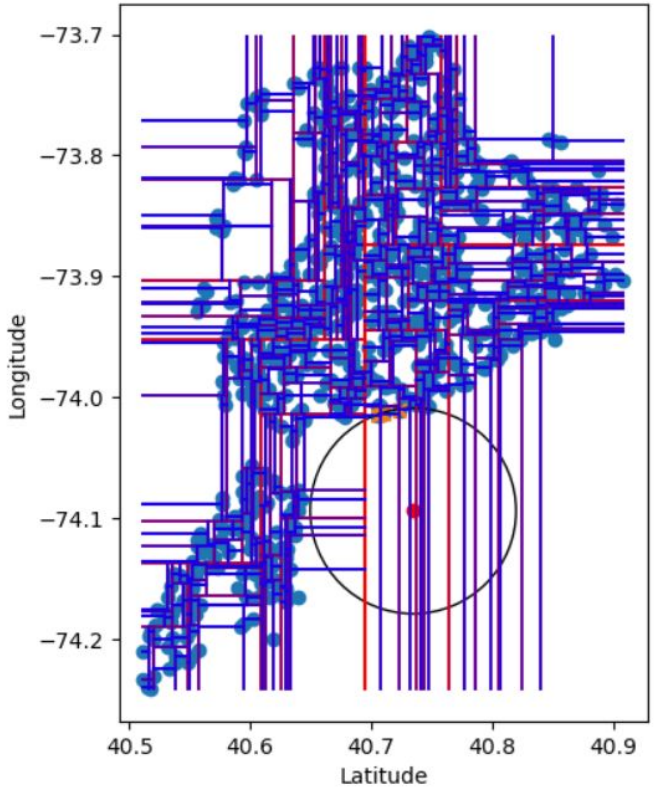
T5 takes prior optimizations and pushes them further with new data structures and algorithmic optimizations. In general, T5 operates under the principle that pre-computation is only needed once, and thus is not a huge detriment. Pre-computed information can be stored, and allow for faster efficiency and better scalability to a larger domain in the simulation phase. As we will see, this is in fact the case, and T5 far exceeds the runtime performance of T3 and T4.

Average Passenger Wait Time	Total Driver Profit	Average Driver Profit	Pre-Computation Runtime	Simulation Runtime
37.175 minutes	42384.267 minutes	84.938 minutes	71.440 seconds	536.348 seconds

T5 - Node Assignment via KD-Tree

The first space optimization is a KD-Tree implementation for matching drivers and passengers to nearest nodes. Using this data structure, we reduce a linear time complexity to logarithmic time complexity.

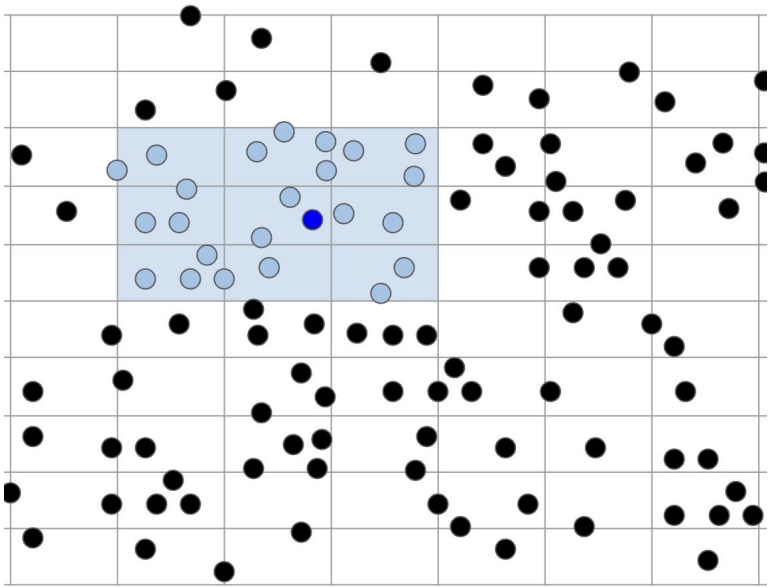
- Whereas the grid-partition system with *M* partitions was expected to maintain the polynomial time complexity of node assignment but reduce it by a constant factor (ND → ND/M, NP → NP/M), the kd-tree reduces the asymptotic complexity to logarithmic (ND → D log N, NP → P log N) by partitioning space such that subpartitions can be represented as a binary tree
- Another major benefit we observe from the data is that we waste less space. The distribution of the points results in many grid spaces being entirely empty. Since a KD-Tree splits space based on the data, we guarantee a more efficient storage and search structure.



T5 - Grid Partitioning for matching

Although we replace grid partitioning for node assignment, we use a variation of the existing grid partitioning as a new system to match drivers and passengers. The grid stores certain information relevant for calculations:

- A dynamically updating list of drivers who are available in a region, including drivers who are about to arrive.
- Average speeds of all roads within a region at a given day and time, weighting each road's contribution to the average based on the amount of road which intersects with the region. This is done via line-segment rectangle intersection calculations.



T5 - Grid Partitioning for matching

Several changes are made to the matching algorithm:

- Matching now takes into account drivers who will arrive to a region in the future, and may arrive closer to the passenger resulting in lower time to reach the passenger. Additionally, it uses the region and time specific average speeds to better approximate driver distances with Manhattan distance.
- We floodfill search the regions, starting from the passenger's coordinates to find the nearest drivers. While this does mean in worst case scenario, we search every region and every driver, we prefer this over the bounds of running Djikstra's or A* multiple times over every node. We achieve O(DM) as opposed to O($DE\log N$) since we use constant time heuristic evaluation instead of graph search. While we have the potential to choose a sub-optimal driver, we drastically improve efficiency and scalability of our algorithm this way (consider for example, if our graph spanned the entire nation, then N may grow linearly but it is likely E grows faster in some combinatorial way).

T5 - Metrics

Although T5 achieves a higher wait time for passengers compared to T3-4, it makes much better profits for drivers. We expect that the wait time should be higher, since T5 often achieves sub-optimal drivers using heuristics as opposed to achieving the exact optimal driver in T3. Since T3/4 calculates graph traversal for all drivers but T5 only selects the closest drivers within a region, we may miss drivers which take fast roads that might be longer as opposed to slow roads that might be closer.

The biggest difference is of course the simulation runtime. We notice that T5 has higher pre-computation runtime because it needs to construct the space partitioning grid and KD-Tree. However, the payoff is magnitudes of speed improvements. Instead of searching all nodes to match our drivers and passengers, we only use logarithmic complexity using KD-Tree. Instead of running several iterations of graph traversal, we only run linearly scaling loops over regions.

B1

While we have taken extensive look at how to weigh drivers for passengers, it would be worth looking at how to weigh passengers themselves. In our situations, we have sorted passengers by arrival time. A potential alternative weighting considers not only the arrival time, but also the distance of the requested ride.

P = (time of arrival) + (distance to destination)

This weights the passengers such that we do consider when they arrive, but additionally provide a penalty on large distance trips so that we are able to circulate nearby drivers more quickly. A driver who only needs to take someone a few blocks down will be able to return and provide another trip, whereas a driver who needs to take someone hours away won't be able to provide any more rides to a location during rush hour.

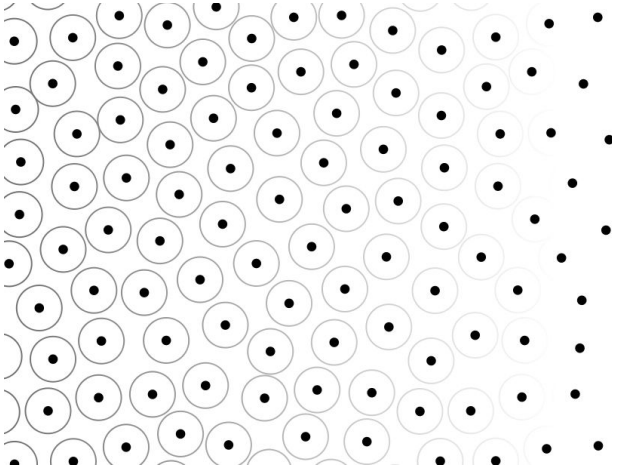
B2

A more future-sighted algorithm for drivers may optimize for *even spread of drivers*. What this means is that ideally, we have drivers everywhere, not clumped up in one spot. This way, in general there will always be a driver close to any passenger, which minimizes the time for a driver to reach a passenger. Additionally, maintaining a good spread of drivers means drivers get an even spread of work, giving each driver a more fair spread of rides.

In practice, this may be achieved by weighing driver choice not only be distance to passenger, but by relative density of drivers in the destination region.

D = (dist to passenger) + (density of drivers at end region) / (estimated density of passengers at end region) * scale

We scale the density of drivers by the density of passengers to ensure that high demand regions receive higher amounts of drivers.



B3

To reduce congestion, we may modify our path search algorithm to propagate a decaying penalty on all traversed paths. When a driver decides on a path, we update our graph weights with a time-stamped penalty such that future graph traversals on that edge will incur a penalty.

Edge Weight = (avg speed) * (length) + (penalty) * f((penalty time) - (current time)) * (scale)

We define f to be a decay function, f(0) = 1 and f(∞) = 0. Using this equation, performing A* or Djikstra's will result in weighing the "busy-ness" on a given road in addition to the time it would take to drive on it, forcing drivers to avoid roads where we know that we have travelled many times.

Each time a driver crosses the edge, we may add onto the penalty, and add a fraction of the time since the previous penalty time to the penalty time to partially refresh the penalty time.

B4

I will incorporate years of historical data to improve our algorithm by first improving the functions that calculate the estimated time for the driver to arrive at the passenger location before a ride, as well as the functions for estimating the time for the passenger to arrive at their final destination. I will use the historical average speed data to analyze how the average speeds over the edges in our graph network have varied over the different days of the week, different months and quarters of the year, and over the past couple of years. I will use this finer granularity of time stamps and average speed data to greatly improve the accuracy of our ETA function. Since our A* algorithm uses the average speed historical data for its heuristic function (since we calculate euclidean distance ÷ average speed), this will lead to a more accurate heuristic function. This improved heuristic functions means that the A* algorithm will exhibit faster convergence and explore fewer nodes to result in improved overall performance for pathfinding. In addition, I will also use historical data trends about traffic conditions, common road closures and construction areas to update and delete some edges in my graph network that correspond to the different streets.