



NotUber Case Study

— Jaque Gomez, Trevon Helm,
Emily Hughes, Celia Vergara —

Executive Summary

Out of the five tasks, T3 (Dijkstra's algorithm) took the longest to complete and had the highest passenger waiting time. The shortest to complete, as expected, was T1, though it had the lowest profit for the drivers.

Though T4 was significantly faster than T3, and had a higher profit than T3. T4 uses A*, which usually, while faster, can also be less accurate in finding the shortest path, which means less profit. T4 did however have a significantly higher driver driving time, which could mean that it was less accurate in finding the shortest path.

Because T3-T5 tried finding the fastest way to the passenger and their drop-off location, the driving time greatly decreased, compared to T1 and T2.

For T5, we chose to only assign rides that would guarantee drivers a positive profit when possible. This means that for lower driving time, we can potentially get higher profits. However, with T5 it appears that many drivers stopped driving, as on average drivers drove less, and thus drivers had to be assigned

Executive Summary: Potential Improvements

- T1/T2: Incorporating the node network into speed calculations to improve accuracy of these in comparison with the rest of the experiments
- T2: Runtime could be improved by using different, more efficient data structures to help save the straight line distances for each driver.
- T3: Improvements to runtime speed of Dijkstra's algorithm can be incorporated to speed up how long the algorithm runs, such as stopping pathfinding early in the case of a distance threshold.
- T5: Further investigation into improving profits overall instead of focusing on a more greedy approach.

General Algorithm Flow

- The pulse of our algorithm is **curr_time**, which increments on each loop of the algorithm by 1 (representing one second).
- Our algorithm will keep running as long as there are passengers who haven't been picked up yet OR there are passengers in the ready queue who haven't been driven yet OR we run out of drivers.
- All algorithms make use of **Driver and Passenger objects**, which handle a lot of the timing of events for use in benchmarking. More details on them in future slides.

General Algorithm Flow (continued)

- If the `curr_time` is greater than or equal to the start time for a passenger/driver, we add them to their respective ready queues
- After this, while there are drivers and passengers in our ready queues, they need to be matched to complete a successful drive. **How a driver/passenger is picked is dependent on each T requirement!**
- Note that at the end of each drive, there is always a chance that a driver will stop driving. To handle this randomness, we check if a random number between 0-1000 is above a threshold (900 unless specified otherwise). If it is, the driver is done driving. Each T may handle a driver leaving differently.
- Rinse and repeat at the top of the loop...

T1

- When it is time for a drive to occur, T1 will simply pop the first passenger and the first driver in our ready queues.
- The node network is not used in T1– as a result, in order to determine the time taken to pickup a passenger, two assumptions are made:
 1. **Haversine Distance:** The distance between the passenger and the driver, along with distance between the passenger and the pickup spot, is calculated using the Haversine Distance. We chose this because it is more accurate when using Latitudes/Longitude coordinates to describe distances due to the geometry of Earth.
 - a. Note: It is not perfect due to the fact that Earth is not a perfect sphere, but we figured it was good enough for our purposes. [Source](#)

T1 (Continued)

2. **Miles per Hour is set to 35 mph.** Since we do not have the speeds of each of the roads, we will assume that all roads are at 35 mph. This follows the law in North Carolina that for residential roads, unless otherwise specified, the maximum speed limit is 35 mph.

- Using both these assumptions, we calculate the time taken to drive passengers in minutes. We then factor that into our profit calculations and time driven for passengers and drivers, along with driver profit calculations. (We also update this for a driver when traversing through the ready queue so that the same driver doesn't repeatedly drive passengers.)

T1 (Numerical Benchmarks)

Runtime (sec)

0.1802225112915039

- Graph comparison of benchmarks is on the final Benchmark slides.

Minimum Passenger Wait Time (min)	Average Passenger Wait Time (min)	Maximum Passenger Wait Time (min)
3.86745972254309e-05	0.03178734840322537	13.990842075355385

Minimum Driver Profit (min)	Average Driver Profit (min)	Maximum Driver Profit (min)
-0.06746885190450094	-0.000502992898154871	0.0509173551238776

Minimum Driver Time (min)	Average Driver Time (min)	Maximum Driver Time (min)
0.023358502777267084	110.37294380901483	944.7146556514292

T2

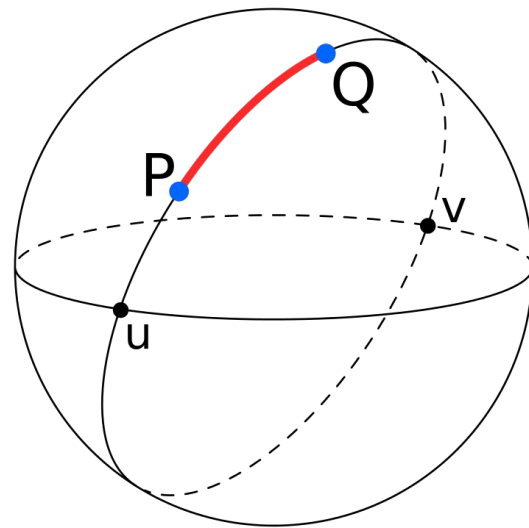
Conceptually very similar to T1, the only difference is we factor in the **minimum straight-line distance** among available drivers.

The algorithm functions the exact same as T1, except we don't pop off the first available driver off the available drivers list. Instead, the steps are:

1. Find the **nearest pick-up point on the road network** to where the passenger is
2. Calculate the **straight-line distance** from the nearest pick-up point to the location of all available drivers
3. Assign the passenger the driver that has the minimum straight-line distance
4. Remove the driver from the available drivers list

T2: Other Considerations - Distance Calculations

- We used **Haversine distance** to calculate straight-line distance instead of the typical Euclidean Distance
- Haversine is more appropriate because distances are described in terms of **latitude and longitude**



T2 (Numerical Benchmarks)

Runtime (sec)

608.3089

- Graph comparison of benchmarks is on the final Benchmark slides.

Minimum Passenger Wait Time (min)	Average Passenger Wait Time (min)	Maximum Passenger Wait Time (min)
0	0.0109	13.99084

Minimum Driver Profit (min)	Average Driver Profit (min)	Maximum Driver Profit (min)
-0.01337	0.032102	0.2115

Minimum Driver Time (min)	Average Driver Time (min)	Maximum Driver Time (min)
0.00033	131.6214	1557.1323

T3

Like T1, the drivers are assigned based to the longest waiting passenger. Except unlike T1 we don't assign the first available driver to the passenger.

Instead, we:

1. Find the **nearest pick-up point** on the road network to where the passenger is
2. Calculate the **shortest path** (the least amount of time it takes to reach the passenger) using Dijkstra's algorithm(modified) for the available drivers
3. Assign the passenger the driver that will reach it the fastest
4. Calculate the **shortest path** (the least amount of time it takes to reach the passengers' drop-off location) starting at the pick-up location to choose what roads to drive on
5. Adjust the start time and location of the driver
6. Remove the driver from the available drivers list
7. Add the driver back to the drivers (all the drivers) list if they aren't done with their shift yet (they still want to continue picking up passengers)

T3- Algorithm Explained

Before any searching begins, the algorithm:

- Initializes an empty dictionary `min_time_to_node` to keep track of the minimum time to reach each node. The starting node (`start_node`) is set to have a minimum time of 0.
- Initializes a priority queue (`priority_queue`) with a tuple `(0, start_node)`, where the first element is the current time and the second element is the starting node.
- Initializes an empty set `visited_nodes` to keep track of visited nodes.

While the priority queue is not empty:

- It pops the node with the minimum time (`current_time`) from the priority queue.
- If the node is already visited, it skips to the next iteration and iff the current node is the destination (`end_node`), the function returns the minimum time to reach the destination.
- If \wedge has not occurred, it explores the neighbors of the current node.
- For each neighbor of the current node it:
 - Retrieves the edge weight (travel time) between the current node and its neighbor by considering the day of the week (weekend/weekday) and the current hour.
 - Calculates the arrival time at the neighbor node (`arrival_time`) by adding the edge weight to the current time.
 - Updates the minimum time to reach the neighbor node if the new arrival time is less than the previously recorded time.
- Pushes the updated information (arrival time and neighbor node) into the priority queue.

T3: Other Considerations

- Like T1:
 - In order to randomly decide when a driver is done driving, we use a **random number generator** for 0 to 1000 and see if it is greater than a threshold of 900. If it is, the driver is done driving. This means that different results are achieved and sometimes passengers are left unassigned(which does happen on ride sharing companies)
 - We save all driven passengers and done drivers to their own list for benchmarking purposes.
- We prioritize the passengers' wait time and don't calculate how long it takes to drop-off the passenger until after the driver is assigned, this means that drivers sometimes take on rides that aren't profitable

T3 (Numerical Benchmarks)

Runtime (sec)

5204.4453

- Graph comparison of benchmarks is on the final Benchmark slides.

Minimum Passenger Wait Time (min)	Average Passenger Wait Time (min)	Maximum Passenger Wait Time (min)
0.9391	638.323	1374.4847

Minimum Driver Profit (min)	Average Driver Profit (min)	Maximum Driver Profit (min)
-75.6873	0.01088549325	87.4450

Minimum Driver Time (min)	Average Driver Time (min)	Maximum Driver Time (min)
2.8887	140.066	785.5493

T4

Our version of T4 takes the baseline code of T3 and improves it in two ways.

1. The implementation of a **2D Binary Search Tree** to store and search coordinates, allowing the finding of closest nodes in sublinear time ($O(\log |V|)$, where V is the number of nodes in the tree). This functionality is leveraged to place our drivers and passengers on the road map accurately and also serves to optimize our pathfinding algorithm by choosing the closest driver first.
 - a. We run the pathfinding algorithm with the closest driver and save that estimated time as the minimum time instead of 'infinity' and when iterating through the rest of the available drivers, we quit attempting to pathfind when the estimated time exceeds the current minimum.
 - b. 2D Binary Search Trees leverage depth to allow for the storage and binary searching of elements with more than one dimension, perfect for coordinates.

T4 Optimization Evidence

```
passenger_1 = passengers[0]
min_dist = float('inf')
closest_node = None

dict_start = time.time()

for key, value in node_map:
    dist = distance.haversine_distance(passenger_1.start_location[0],
                                       passenger_1.start_location[1], key, value)
    if dist < min_dist:
        min_dist = dist
        closest_node = node_map[key, value]

dict_end = time.time() - dict_start

print("Time to run brute force calculation:", dict_end)
```

```
tree_start = time.time()

closest_node_tree = search(tree_root, passenger_1.start_location)

tree_end = time.time() - tree_start

print("Time to run tree search:", tree_end)

if closest_node == closest_node_tree.id:
    print("Nodes are equivalent!")
else:
    print("Nodes are not equivalent!")
```

Time to run brute force calculation: 0.18552303314208984

Time to run tree search: 0.0

Nodes are equivalent!

Linear vs. Sublinear Performance

T4 Efficiency With and Without Driver Choice Optimization

Without:

```
Passenger: 200 Estimated Time: 5.214983
--- 73.54880785942078 seconds ---
Replacing closest driver! 5
Replacing closest driver! 5 # of available drivers
Passenger: 201 Estimated Time: 18.452796
--- 74.71233797073364 seconds ---
Replacing closest driver! 4
Replacing closest driver! 4
Replacing closest driver! 4
Passenger: 202 Estimated Time: 28.061551
--- 75.76415348052979 seconds ---
Replacing closest driver! 7
Replacing closest driver! 7
Replacing closest driver! 7
Replacing closest driver! 7
Replacing closest driver! 7
```

1. 73 seconds @ 200 passengers - INEFFICIENT
2. Replacing closest driver SEVERAL times

With:

```
Passenger: 200 Estimated Time: 5.214983
--- 41.40330457687378 seconds ---
Replacing closest driver! 13
Passenger: 201 Estimated Time: 11.805308
--- 42.1205313205719 seconds ---
Passenger: 202 Estimated Time: 1.029784
--- 42.14588522911072 seconds ---
Replacing closest driver! 14
Passenger: 203 Estimated Time: 6.682988
--- 42.39657258987427 seconds ---
Replacing closest driver! 13
```

1. 41 seconds @ 200 passengers - EFFICIENT
2. Replacing closest driver only once or not at all

- Note that the WITH algorithm is dealing with 13-14 available drivers while WITHOUT is only dealing with 4-7!

T4: Optimization 2

T3 Run - Dijkstra's

```
Passenger: 100 Estimated Pickup Time: 0.528608 Estimated drop-off time 21.362562  
--- 254.43413281440735 seconds ---  
Passenger: 101 Estimated Pickup Time: 2.730842 Estimated drop-off time 8.734775  
--- 257.7170696258545 seconds ---  
Passenger: 102 Estimated Pickup Time: 1.674156 Estimated drop-off time 6.672933  
--- 260.95897579193115 seconds ---
```

A* alone improves the runtime efficiency of our algorithm by a factor of over 2.

* This T4 run does not include driver optimization or early algorithm stoppage for the most accurate Dijkstra's vs. A* runtime.

T4 Run* - A*

```
Passenger: 100 Estimated Time: 0.528608  
--- 99.60749697685242 seconds ---  
Passenger: 101 Estimated Time: 2.730842  
--- 100.8765139579773 seconds ---  
Passenger: 102 Estimated Time: 1.235850  
--- 102.30904054641724 seconds ---
```

2. Our pathfinding algorithm is improved from Dijkstra's algorithm to the A* algorithm, using the straight-line distance (Haversine distance) as a heuristic for choosing the most likely shortest paths.

T4 (Numerical Benchmarks)

Runtime (sec)

1448.012834072113

- Graph comparison of benchmarks is on the final Benchmark slides.

Minimum Passenger Wait Time (min)	Average Passenger Wait Time (min)	Maximum Passenger Wait Time (min)
0.0	8.018840562550098	81.22148867123632

Minimum Driver Profit (min)	Average Driver Profit (min)	Maximum Driver Profit (min)
-49.22036682559198	50.29615664666197	395.88661904561616

Minimum Driver Time (min)	Average Driver Time (min)	Maximum Driver Time (min)
3.453119172226633	308.42765668542785	2825.3655464037815

T5

Like T4, except:

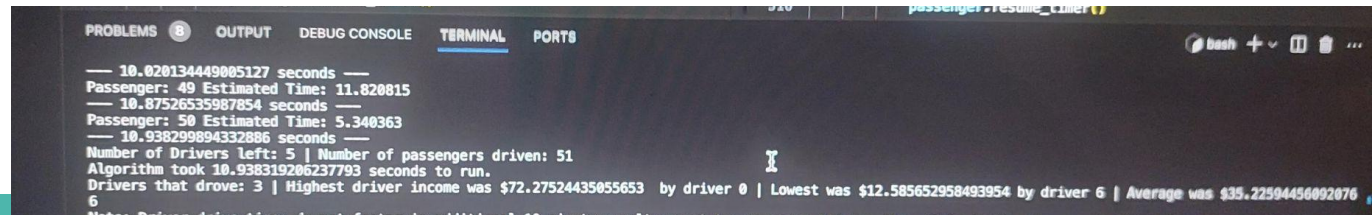
- For each possible driver we calculate the profit $[(\text{drop-off distance}) - (\text{pick-up distance})]$ that they would get from picking up that passenger to try and assign only rides that have a positive profit, when possible.

T5: Other Considerations

- Like T1:
 - In order to randomly decide when a driver is done driving, we use a **random number generator** for 0 to 1000 and see if it is greater than a threshold of 900. If it is, the driver is done driving.
 - We save all driven passengers and done drivers to their own list for benchmarking purposes.
- Unlike T3- T4, we only assign rides that are profitable
 - In order to do so we calculate the profit a driver will get when assigning them to a passenger

T5 Optimizations Evidence

- As will be shown in the graphs later on in this slideshow, our plan to improve driver profit backfired in a fascinating way: our lowest driver profit plummeted, as did our average profit. Our highest driver profit stayed similar to T4 results. However, our runtime improved and our highest driving time was lowered significantly!
- This is a case where a greedy algorithm was not quite the best algorithm– it did work, but more work can be done for an overall more stable profit margin than a wilder one.
- Our algorithm may have failed to increase profit because drivers that were close to the passenger but far enough away to not make a profit would get overlooked in favor of drivers that would make a profit and then were only scheduled when the driver pool shrank and no profitable drives were possible
- This is supported by runs on smaller datasets in which profit did increase. It seems that as demand increases, this algorithm begins to fail.



```
PROBLEMS 8 OUTPUT DEBUG CONSOLE TERMINAL PORTS
— 10.020134449005127 seconds —
Passenger: 49 Estimated Time: 11.820815
— 10.87526535987854 seconds —
Passenger: 50 Estimated Time: 5.340363
— 10.938299894332086 seconds —
Number of Drivers left: 5 | Number of passengers driven: 51
Algorithm took 10.938319206237793 seconds to run.
Drivers that drove: 3 | Highest driver income was $72.27524435055653 by driver 0 | Lowest was $12.585652958493954 by driver 6 | Average was $35.22594456092076
6
```

T5 (Numerical Benchmarks)

Runtime (sec)

600.7735800743103

- Graph comparison of benchmarks is on the final Benchmark slides.

Minimum Passenger Wait Time (min)	Average Passenger Wait Time (min)	Maximum Passenger Wait Time (min)
0	69.71384088355565	432.8081014719144

Minimum Driver Profit (min)	Average Driver Profit (min)	Maximum Driver Profit (min)
-149.82984507509502	19.128142048155237	363.7613374622826

Minimum Driver Time (min)	Average Driver Time (min)	Maximum Driver Time (min)
3.7021079935698316	233.85918220792908	1654.1444454099587

B1

Some ideas to deal with a surge situation are:

- Creating a **carpool system** where drivers can pick up more than 1 passenger if they are generally on the way
 - This would allow less drivers to pick up more passengers, potentially minimizing wait time for passengers
- Allow and prioritize **“advanced booking”**
 - If a passenger requested a ride earlier than others and have scheduled it, prioritize minimizing their waiting time if they have planned it ahead
 - Given a driver shortage, this seems like an equitable way to prioritize passengers

B2

Some ways we could improve driver profit without hurting passenger wait time:

- Bonuses for drivers who drive during peak hours or for drivers who choose to complete x amount of rides in a specific timeframe
- Incorporating a tip system to the algorithm- if a passenger is driven under a certain time frame, the driver can receive extra money from the passenger.
- Better schedule drivers throughout the day. If there aren't many passengers during a certain time frame, we can recommend the driver to wait until there is more activity.

B3

If road congestion is becoming an issue...an option is to **add capacities** to the road network detailing the maximum number of drivers that can be sent out on each edge/road.

You send the drivers down the best route unless the max capacity has been reached - then you disregard the edge if that specific one has reached max capacity.

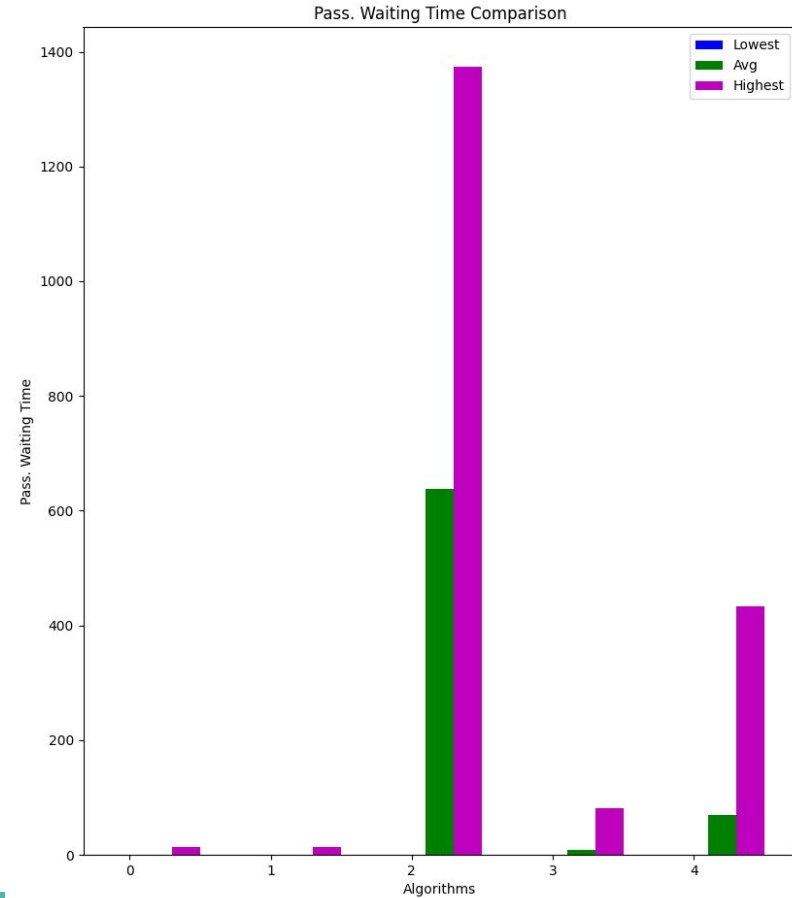
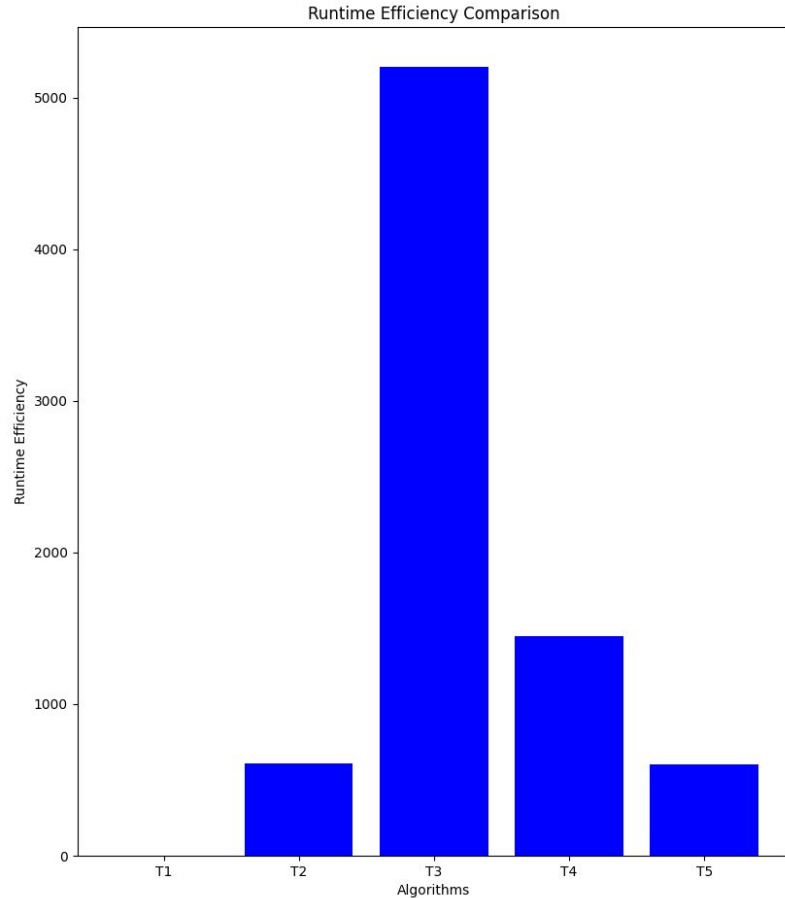
With a capacity argument, we could then make use of Max-Flow algorithms to best send drivers on roads that are less congested.

B4

Some important things that can be done to incorporate historical data to fulfill the desiderata:

- **Pre-positioning** drivers in locations which are more likely to pick up passengers, decreasing wait times for passengers
- **Using A* with improved heuristics**, factoring in historical traffic data to create better estimates of how many minutes it will take to reach the drop-off location

Benchmarks (Efficiency and Passenger Waiting Time for T1-T5)



Benchmarks (Profit and Driving Time for T1-T5)

