Case Study Report

By: Jeff Jung, Claire Tan, Celina You

Executive Summary

Primary Objective: Develop a scalable, efficient algorithm for matching drivers to passengers in a ride-sharing network.

Key Challenge: Minimizing passenger waiting time and maximizing driver profit while handling a large road network efficiently.

Summary of our findings:

- **Initial Approach with Dijkstra's Algorithm:** Started with a basic implementation using Dijkstra's algorithm for pathfinding between nodes, providing accurate but time-intensive solutions.
- **Introduction of Binary Search Trees (BSTs):** Incorporated BSTs for efficient querying of the nearest nodes, reducing the computational load compared to naive approaches.
- **Transition to k-d Trees:** Shifted from BSTs to k-d Trees for more efficient spatial searches, significantly improving the matching of drivers to passengers based on proximity.
- **Implementation of A Search Algorithm:*** Integrated A* search for pathfinding, combining it with k-d Trees for more efficient navigation. This allowed for the rapid calculation of optimal paths, taking into account both geographic constraints and real-time traffic conditions.
- **Optimization with Geohashing:** Evolved to using geohashing, enabling faster geographical searches and efficient clustering of nodes, which led to rapid identification of eligible passengers for drivers.

T1

Objective: Match drivers with passengers based on their arrival times, aiming to minimize passenger waiting time (D1) and maximize driver profit (D2)

Algorithm:

1. Sort both drivers and passengers in ascending order based on their 'Date/Time' to process them chronologically.
2. Initialize two queues: one for drivers and another for passengers. Enqueue all passengers.
3. Driver-Passenger Matching:
    - Iterate over each driver in the sorted list.
    - For each driver, dequeue a passenger from the passenger queue (if available).
    - Ensures each driver was matched with the next available passenger
    - Calculate the waiting time for the passenger (difference in minutes between the driver's and the passenger's 'Date/Time')

D1: Average Waiting Time

Objective: Evaluate the average waiting time for passengers when matched with drivers.

Experiment Methodology::

- Processed real-world passenger data to simulate request times
- Matched each passenger with the next available driver based on time order
- Calculated waiting time as the time difference between passenger request and driver availability
- Total waiting time for all passengers was aggregated - providing a fair representation of the overall passenger experience across the system

Results:

- Average Waiting Time: **120.56 minutes**

Analysis:

- The high average waiting time indicates a need for improvement in reducing passenger wait times
- Suggests exploring more efficient matching algorithms or prioritization strategies

D2: Average Driver Profit

Objective: Assess the average profit for drivers under the current matching algorithm.

Experiment Methodology::

- Utilized chronological matching of drivers with passengers
- Simplistically calculated driver profit based on passenger waiting time as a placeholder
- Driver profit = time drivers spend transporting passengers
- Total profit for all drivers was aggregated - providing a fair representation of the overall driver experience across the system

Results:

- Average Driver Profit: **1208.53 minutes** (as per current profit model)

Analysis:

- The profit calculation model is currently simplistic and may not accurately reflect real-world earnings
- Recommends developing a more realistic profit model for future evaluations

D3: Efficiency & Scalability

Objective: Evaluate the efficiency and scalability of the algorithm based on its execution time and resource usage

Experiment Methodology::

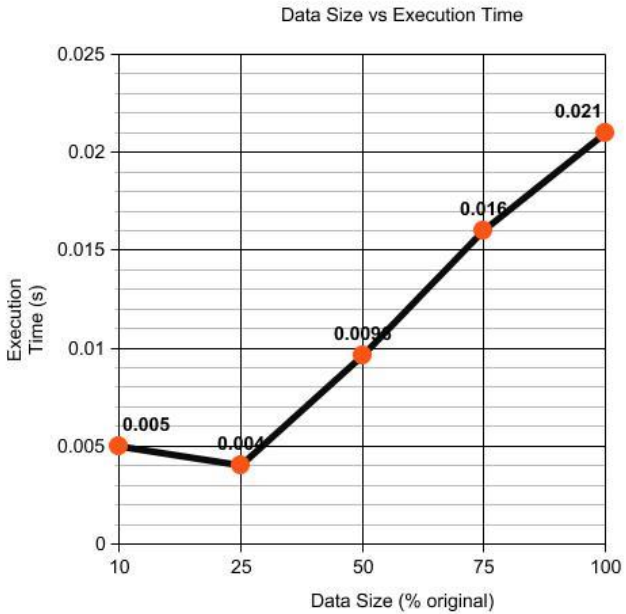
- Create multiple subsets of original driver/passenger data, each of increasing size (10%, 25%, 50%, 75%, 100%)
- Run the algorithm on each subset and record the runtime

Results:

- Empirical Runtime: **0.021 seconds**

Analysis:

- The T1 algorithm is efficient for small to medium datasets, as evidenced by the quick runtime. It's a good starting point for a basic pairing system.
- Scalability for very large datasets or more complex pairing criteria may be limited. Improvements like memory optimization, parallel processing, and more sophisticated data structures could enhance scalability.



T2

Objective: Improve upon the baseline matching algorithm by considering the geographical proximity of drivers to passengers, still aiming to minimize passenger waiting time (D1) and maximize driver profit (D2)

Algorithm:

1. Sort drivers and passengers based on their availability times
2. Implement the Haversine formula to calculate the straight-line distance between two latitude/longitude points
3. Initialize two queues: one for drivers and another for passengers.
4. Match each driver with the nearest available passenger based on geographical proximity
5. Calculate time and distance metrics
    a. Calculates wait time for each passenger as the difference between the passenger's request time and the matched driver's availability time
    b. Computes driving time based on the distance between the passenger pickup and drop-off locations and an assumed average speed (30 MPH)

D1: Average Waiting Time

Objective: Evaluate the average waiting time for passengers using the improved matching algorithm

Experiment Methodology::

- Passengers matched with the nearest available drivers based on geographical proximity
- Wait time calculated as the difference between passenger request time and driver availability time
- Total waiting time for all passengers was aggregated - providing a fair representation of the overall passenger experience across the system

Results:

- Average Waiting Time: **120.56 minutes**

Analysis:

- Despite the proximity-based matching, the waiting time remains relatively high (same as T1)
- Although T2 calculates the geographical distance between pickup and drop-off points, this information is not used to influence the pairing decision. It only affects the calculation of the drive time, not the wait time.
- Indicates potential areas for further optimization, such as dynamic routing or more efficient matching strategies.

D2: Average Driver Profit

Objective: Assess the average driving time/profit for drivers, indicating driver utilization and efficiency.

Experiment Methodology::

- Calculated driving time based on the distance between passenger pick-up and drop-off points, using an average speed of 30 MPH
- Matched drivers with the closest passengers to potentially reduce travel time to pick-up points
- Driver profit = time drivers spend transporting passengers
- Total profit for all drivers was aggregated - providing a fair representation of the overall driver experience across the system

Results:

- Average Driver Profit: **8.01 minutes**

Analysis:

- The lower average drive time suggests improved driver utilization compared to the baseline.
- Suggests that drivers spend less time traveling empty, enhancing overall system efficiency.

D3: Efficiency & Scalability

Objective: Evaluate the efficiency and scalability of the algorithm based on its execution time and resource usage

Experiment Methodology::

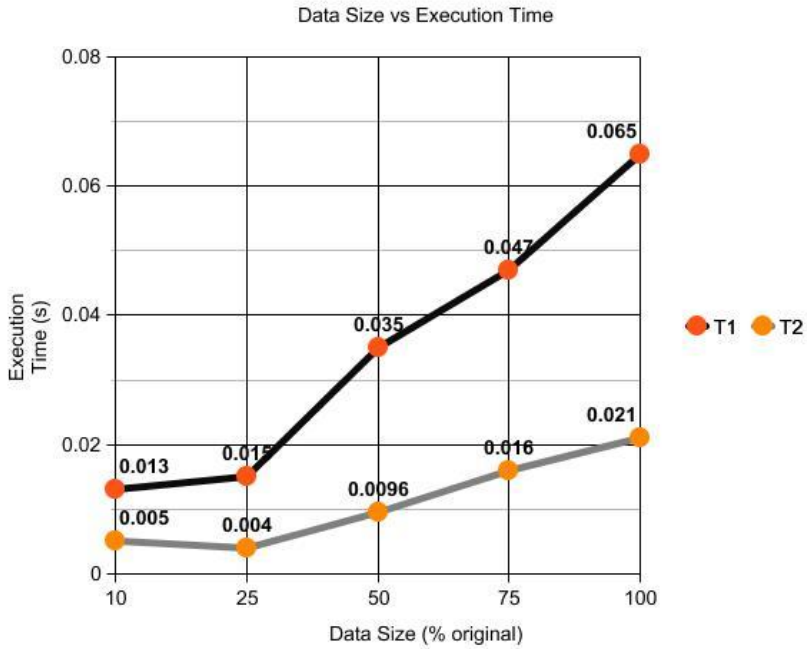
- Create multiple subsets of original driver/passenger data, each of increasing size (10%, 25%, 50%, 75%, 100%)
- Run the algorithm on each subset and record the runtime

Results:

- Empirical Runtime: **0.0645 seconds**

Analysis:

- Based on our experiments, the T2 algorithm is noticeably more efficient that our T1 algorithm, with the T2 runtime for each data size being faster than that of T1.



T3

Objective: Improve upon the geographical proximity approach in T2 to minimize passenger waiting time and maximize driver profit by considering the actual road network for matching

Algorithm:

1. Sort drivers and passengers based on their start times using priority queues.
2. Utilize a Binary Search Tree for efficient location searching based on latitude and longitude.
3. For each driver in the queue, identify eligible passengers whose request time is before or equal to the driver's start time.
4. Implement Dijkstra's algorithm to calculate the shortest path and time for the driver to reach each eligible passenger, considering the current road network.
5. Select the passenger for whom the driver's travel time to reach is the minimum, optimizing for the shortest time to passenger pickup.
6. Calculate the time metrics: waiting time for the passenger and total driving time
7. If the driver chooses to continue after a drop-off, update their location and start time, and re-insert them into the driver queue. Passengers are removed from the queue once matched.
8. Repeat the process until all passengers are matched or no drivers are left in the queue.

D1: Average Waiting Time

Objective: Evaluate the average waiting time for passengers using the improved matching algorithm

Experiment Methodology::

- The total wait and travel time is aggregated across all passengers to assess the efficiency of the algorithm in minimizing passenger wait times.

Results:

- Total Waiting Time: 79006.14 minutes
- Waiting Time Per Passenger: 79006.14 min / 5003 passengers = **15.79 min/passenger**

Analysis:

- An average waiting time of 15.79 minutes per passenger, in the context of urban ride-hailing services, can be perceived as moderately high, especially in a busy city setting where quick service is often a priority for users.
- The use of BST and standard pathfinding may not be the most efficient for a dynamic and complex urban environment, which could lead to the observed average waiting time.
- While BSTs are efficient for sorted data, they might not always provide the fastest lookup for spatial data.
- Dijkstra's can also be slower compared to more optimized algorithms like A*, especially in large, complex road networks
- Further optimizations with KD-Trees and A* algorithm, can potentially reduce this waiting time

D2: Average Driver Profit

Objective: Assess the average profit for drivers under the current matching algorithm.

Experiment Methodology::

- Profit for each driver is calculated as the time spent driving passengers minus the time spent reaching them
- The total driver ride profit is aggregated across all drivers to evaluate how well the algorithm maximizes active, revenue-generating driver time

Results:

- Total Driver Profit: **40,896.70 minutes**
- Profit Per Driver: 40,896.70 min / 500 drivers = **81.79 min/driver**

Analysis:

- An average profit of 81.79 minutes per driver suggests that, on average, each driver spends about 1 hour and 22 minutes in paid travel time more than the time spent reaching passengers
- Our T3 algorithm, which uses a Binary Search Tree for spatial queries and a standard pathfinding approach, might not be the most optimized for minimizing drivers' unpaid travel time
- The T3 algorithm's performance implies that implementing more efficient spatial data structures (like KD-Trees) and pathfinding algorithms (like A*) could significantly improve driver profitability by reducing the time taken to reach passengers

D3: Efficiency and Scalability

Objective: Evaluate the efficiency and scalability of the algorithm based on its execution time and resource usage

Experiment Methodology::

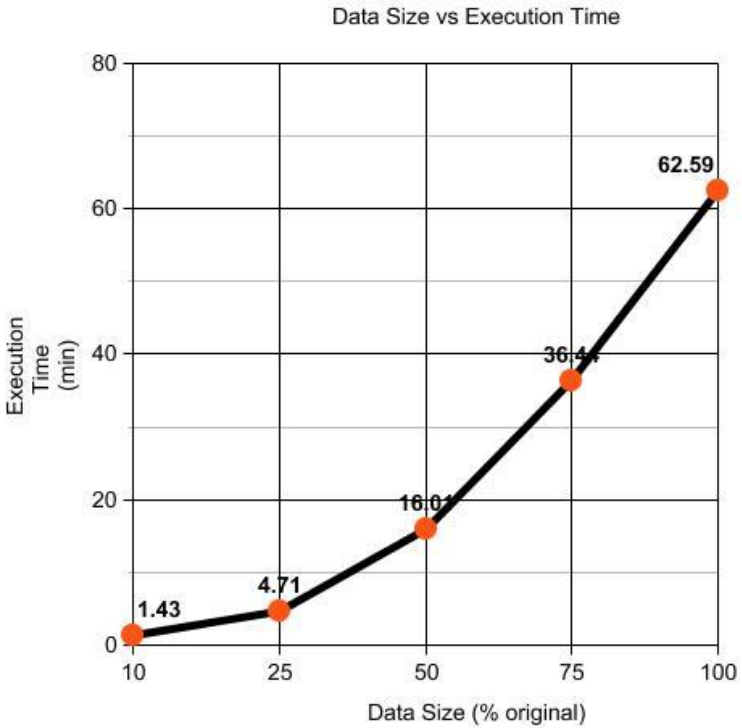
- Create multiple subsets of original driver/passenger data, each of increasing size (10%, 25%, 50%, 75%, 100%)
- Run the algorithm on each subset and record the runtime

Results:

- Empirical Runtime: **62.59 minutes**

Analysis:

- The runtime increases more than linearly with the size of the dataset. This is evident from the fact that doubling the data size from 25% to 50% and then 50% to 100% more than doubles the runtime, suggesting that the algorithm may not scale efficiently with larger datasets
- The T3 algorithm likely involves operations with higher time complexities, such as pathfinding for each driver-passenger pair, which can exponentially increase with the data size
- The use of a Binary Search Tree for spatial querying might not be the most efficient for large, non-uniform datasets
- The data indicates a need for further optimization to handle large datasets efficiently



T4

Objective: Improve the runtime efficiency while maintaining the same general scheduling strategy as in T3

- Preprocess the nodes of the road network so to efficiently find the closest node to a given query point

Algorithm Optimizations:

1. **KD-Tree Construction for Efficient Closest Node Finding:**
   - KD-Tree is constructed using the given node data (latitude and longitude of road network vertices)
   - The purpose of KD-Tree is to efficiently find the closest road network node to any given point (like passenger or driver locations)
2. **Efficient Nearest Neighbor Search:**
   - The KD-Tree is used to efficiently find the nearest node to a given latitude and longitude
   - This is a significant optimization over looping over all nodes in the road network, reducing the search from linear to logarithmic time complexity in average cases
3. **Integration into the Ride Matching Process:**
   - When pairing drivers with passengers, the KD-Tree is used to find the closest network node to both drivers and passengers quickly
   - This optimization helps in quickly identifying the most suitable drivers for passengers based on proximity
4. **Preprocessing for Sub-linear Efficiency:**
   - The KD-Tree construction, where each decision at a node halves the search space, leads to logarithmic time complexity for nearest-neighbor searches
   - This significantly reduces the time complexity from O(n) for linear searching to O(log n) on average for spatial data, where n is the number of nodes in the road network

T4

Objective: Improve the runtime efficiency while maintaining the same general scheduling strategy as in T3

- Compute shortest paths between two nodes of the road network in a way that is more efficient than just Dijkstra's algorithm

Algorithm Optimizations:

1. **Travel Times Precomputation:**
   - New function precompute_travel_times calculates the travel time for each road segment based on length and speed, considering different hours of the day and types of days (weekdays or weekends)
   - Precomputed travel times are stored in a dictionary, enabling quick retrieval during pathfinding computations
2. **A* Algorithm for Efficient Pathfinding*:**
   - A* is an informed search algorithm that uses a heuristic to estimate the cost to reach the goal from a given node, improving efficiency over uninformed search algorithms like Dijkstra's
   - The heuristic used in the A* algorithm is the Euclidean distance between nodes, more efficient compared to calculating the exact path cost at every step, as in Dijkstra's algorithm
   a. The integration of KD-Tree for nearest node lookup and the A* algorithm for efficient pathfinding allows the system to quickly identify the closest road network nodes to drivers and passengers and then efficiently compute the shortest path between these points
3. **Dynamic Graph Construction for Time-Dependent Travel Times:**
   - New function construct_optimized_graph function dynamically constructs the graph of the road network based on the current time
   - Takes into account the variation in travel times at different hours and on different days (weekdays vs weekends), allowing the pathfinding algorithm to use the most relevant travel times

D1: Average Waiting Time

Objective: Evaluate the average waiting time for passengers using the improved matching algorithm

Experiment Methodology::

- The total wait and travel time is aggregated across all passengers to assess the efficiency of the algorithm in minimizing passenger wait times.

Results:

- Total Waiting Time: **74187.11 minutes**
- Waiting Time Per Passenger: 74187.11 minutes / 5003 passengers = **14.83 min/passenger**
- T4-T3 Difference: 15.79-14.83 = **0.96 minutes**

Analysis:

- The decrease in average time per passenger from T3 to T4 demonstrates the effectiveness of the optimizations introduced in T4. Specifically, the use of KD-Trees for efficient spatial querying and A* algorithm for optimized pathfinding contributes to this improvement.
- While a reduction of around 0.96 minutes might seem modest, it's important to consider this in the context of a large-scale urban environment like New York City.
- Although there's a clear improvement, exploring further optimizations or fine-tuning existing parameters could potentially yield even better results. For instance, tweaking the KD-Tree's depth or exploring different heuristics for the A* algorithm might offer additional benefits

D2: Average Driver Profit

Objective: Assess the average profit for drivers under the current matching algorithm.

Experiment Methodology::

- Profit for each driver is calculated as the time spent driving passengers minus the time spent reaching them
- The total driver ride profit is aggregated across all drivers to evaluate how well the algorithm maximizes active, revenue-generating driver time

Results:

- Total Driver Profit: **47974.41minutes**
- Profit Per Driver: 47974.41 min / 500 drivers = **95.95 min/driver**
- T4-T3 = 95.95−81.79 = **14.16 min/driver**

Analysis:

- The increase in average driver profit from T3 to T4 indicates that the optimizations in T4 effectively reduced the time drivers spend traveling to pick up passengers, relative to the time spent on paid rides
- T4 uses a KD-Tree for efficient spatial querying, enabling quicker identification of the nearest driver to a passenger. This likely contributed to reducing the travel time to passenger locations
- The A* algorithm in T4, being more efficient than the standard pathfinding used in T3, likely found shorter or faster routes, further reducing unpaid travel time and hence increasing profit

D3: Efficiency and Scalability

Objective: Evaluate the efficiency and scalability of the algorithm based on its execution time and resource usage

Experiment Methodology::

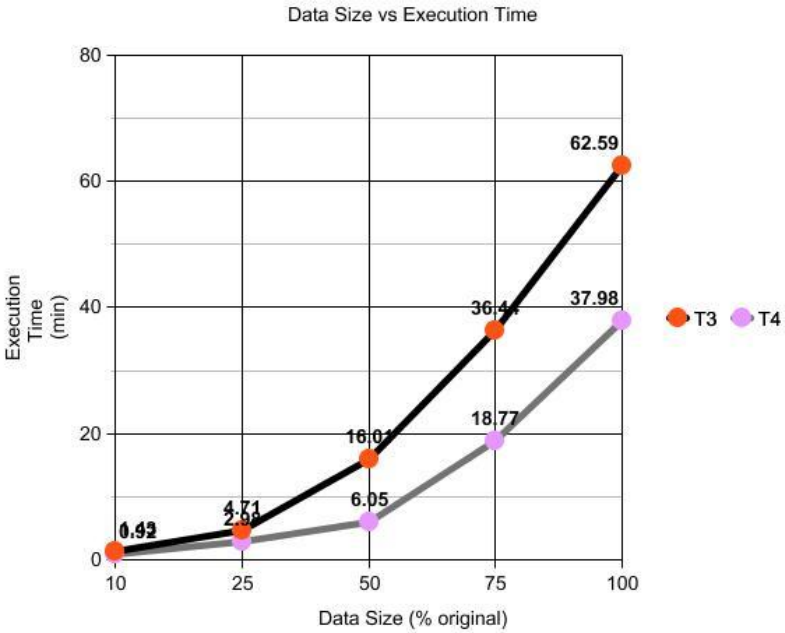
- Create multiple subsets of original driver/passenger data, each of increasing size (10%, 25%, 50%, 75%, 100%)
- Run the algorithm on each subset and record the runtime

Results:

- Total Empirical Runtime: **37.98 minutes**

Analysis:

- At each data size percentage, T4 shows a notable reduction in runtime compared to T3. This is indicative of the efficiency gains achieved through the algorithmic improvements in T4
- Both T3 and T4 exhibit non-linear scaling of runtime with increasing data size. However, the rate of increase in runtime is less steep in T4, suggesting better scalability
- For example, the runtime for 100% of the data in T4 (37.98 minutes) is significantly lower than in T3 (62.59 minutes), showing a marked improvement in handling large datasets
- Despite improvements, the runtimes in T4 still show non-linear growth. This suggests that while T4 is more efficient than T3, further optimizations might be necessary for massive datasets or extremely time-sensitive applications



T5

Objective: Describe and implement your own alternative scheduling algorithm to improve over the baseline algorithms

Algorithm:

1. Geohash Conversion:
    a. Convert the latitude and longitude coordinates of each node in the road network into a geohash string with specified precision.
    b. Create an index (dictionary) where each unique geohash string maps to a list of node IDs that share the same geohash.
2. For each driver, create a HeapElement containing their next available time and information, and insert these elements into a priority queue.
3. Sort all passengers in ascending order based on their requested time, ensuring those who requested a ride earlier are considered first.
4. Construct a graph representing the road network where each edge's weight is the travel time calculated based on current traffic conditions (time of day, weekday/weekend).
5. Driver-Passenger Pairing Process:
    a. For each driver in the priority queue, find the closest node using the geohash index.
    b. For each passenger waiting for a ride, also find the closest node using the geohash index.
6. Eligible Passenger Selection:
    a. For the current driver, evaluate each waiting passenger to determine if they are within an acceptable travel time threshold (NEARBY_TIME_THRESHOLD).
    b. Select the passenger who has been waiting the longest within this threshold.
7. Route Calculation:
    a. For the selected driver-passenger pair, calculate the shortest path and travel time from the driver's current location to the passenger's pickup location and then to their drop-off location.

D1: Average Waiting Time

Objective: Evaluate the average waiting time for passengers using the improved matching algorithm

Experiment Methodology::

- The total wait and travel time is aggregated across all passengers to assess the efficiency of the algorithm in minimizing passenger wait times.

Results:

- Total Waiting Time: **72889.04 minutes**
- Waiting Time Per Passenger: 72889.04 minutes / 5003 passengers = **14.57 min/passenger**
- T5-T3 Difference: 15.79-14.83 = **0.96 minutes**
- T5-T4 Difference: 14.83-14.57 = **0.26 minutes**

Analysis:

- The reductions in waiting time per passenger in T5, compared to both T3 and T4, suggest that the algorithmic changes made in T5, particularly the introduction of geohashing, contribute positively to operational efficiency
- Geohashing likely enables faster and more accurate matching of drivers to passengers based on geographical proximity, which can lead to reduced waiting times
- The difference in waiting times between T5 and T4 is smaller than that between T5 and T3, indicating that while T5 continues to improve upon the previous algorithms, the margin of improvement is narrowing. This suggests that T5 is approaching an optimized state for this particular metric.

D2: Average Driver Profit

Objective: Assess the average profit for drivers under the current matching algorithm.

Experiment Methodology::

- Profit for each driver is calculated as the time spent driving passengers minus the time spent reaching them
- The total driver ride profit is aggregated across all drivers to evaluate how well the algorithm maximizes active, revenue-generating driver time

Results:

- Total Driver Profit: **51787.90 minutes**
- Profit Per Driver: **51787.90** min / 500 drivers = **103.58 min/driver**
- T4-T3 = 103.58−81.79 = **21.79 min/driver**
- T5-T4 = 103.58-95.95 = **7.63 min/driver**

Analysis:

- The progression from T3 through T4 to T5 shows a consistent increase in driver profit, indicating successive improvements in the efficiency of driver-passenger matching and route optimization
- The implementation of geohashing in T5 might have contributed to quicker and more efficient driver-passenger pairings, reducing the time drivers spend in transit to passengers, thereby increasing their time on paid rides
- The gain from T4 to T5, while smaller than from T3 to T4, is still significant, especially considering the already optimized state of T4. It indicates that T5's changes, while seemingly incremental, have a meaningful impact on operational efficiency

D3: Efficiency and Scalability

Objective: Evaluate the efficiency and scalability of the algorithm based on its execution time and resource usage

Experiment Methodology::

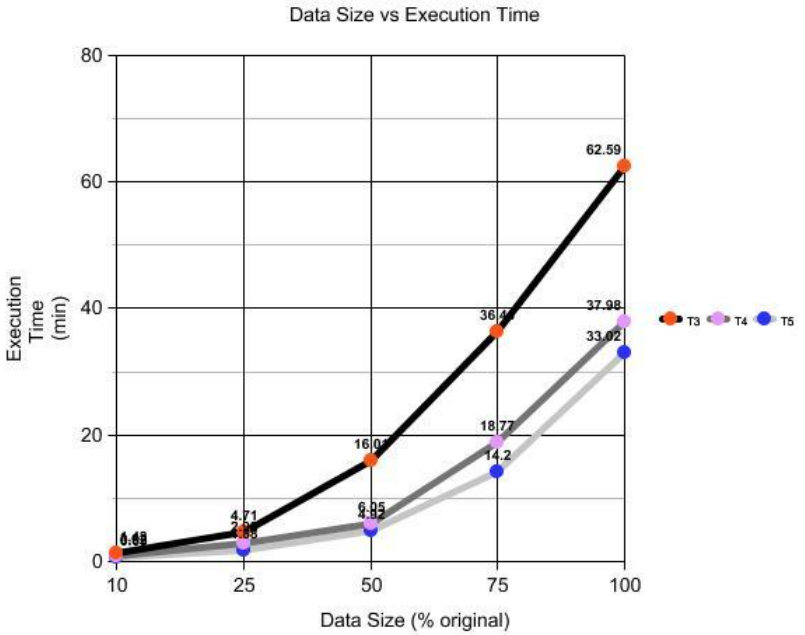
- Create multiple subsets of original driver/passenger data, each of increasing size (10%, 25%, 50%, 75%, 100%)
- Run the algorithm on each subset and record the runtime

Results:

- Total Empirical Runtime: **37.98 minutes**

Analysis:

- Comparing the D3 results of T5 with those of T3 and T4 demonstrates clear improvements in processing efficiency and scalability in T5 through the implementation of geohashing
- T5 scales better with increasing data size than T3 and T4. The non-linear growth in runtime is still present but is less pronounced in T5, indicating a more scalable solution for larger datasets
- T5's flexibility in adjusting geohash precision and similarity thresholds likely contributes to its performance, balancing the trade-off between spatial accuracy and computational demand
- This suggests that T5 is a more viable and efficient solution for real-time operations in large-scale ride-hailing services



B1

Objective: Modify our algorithms to cope with high demand while maintaining fairness

2. Geographic Distribution of Drivers

a. **Strategy**: Distribute drivers geographically to cover high-demand areas more effectively.
b. **Algorithm Adjustment**: Analyze demand in different geographic zones and direct drivers accordingly.
c. **Implementation**:
    i. Divide the service area into zones and track demand in real-time per zone
    ii. Use a clustering algorithm (like K-means) to identify high-demand areas
    iii. Use incentives or notifications to encourage drivers to move towards high-demand zones.
    iv. When a request comes in, assign the nearest available driver
d. **Fairness Aspect:** Aims to reduce wait times by having drivers more evenly distributed, catering to passengers across different areas. However, this might disadvantage passengers in less populated areas who might experience longer wait times

B1

Objective: Modify our algorithms to cope with high demand while maintaining fairness

3. Surge Pricing

a. **Strategy**: Implement surge pricing to manage demand; higher fares can incentivize more drivers to serve in these areas while potentially moderating the number of ride requests.
b. **Algorithm Adjustment**: Implement dynamic pricing based on real-time supply and demand
c. **Implementation**:
    i. Monitor the ratio of available drivers to ride requests in real-time
    ii. Temporarily Increase prices in areas with higher demand-to-supply ratios to balance the demand
    iii. Surge pricing can be dynamic, increasing as demand increases and decreasing as it normalizes
d. **Fairness Aspect**: While surge pricing may affect affordability, it can help balance supply and demand, reducing wait times overall. However, it might be perceived as unfair by passengers who cannot afford higher prices during peak times

B1

Objective: Modify our algorithms to cope with high demand while maintaining fairness

4. Predictive Analysis for Demand Forecasting

a. **Strategy**: Use historical data to proactively position drivers in high-demand areas at the right times, based on trends and patterns observed in the past
b. **Algorithm Adjustment**: Aggregate and train a model on historical data to anticipate future demand patterns
c. **Implementation**:
    i. Use historical data to predict demand patterns using machine learning models (like time series forecasting).
    ii. Position drivers in anticipation of high demand in specific areas and times.
    iii. Update these predictions regularly based on the latest data
d. **Fairness Aspect**: Predictive positioning of drivers can ensure quicker service during peak times. However, there is a risk of over or under-estimating demand in certain areas, leading to potential imbalances

B1

Objective: Modify our algorithms to cope with high demand while maintaining fairness

5. Load Balancing Among Drivers

a. **Strategy**: Ensure a fair distribution of ride requests among drivers, balancing their workloads and earnings opportunities
b. **Algorithm Adjustment**: Use a queue or scoring system to prioritize drivers who have had fewer trips or less working time when a ride request is made
c. **Implementation**:
    i. Track the number of rides or working hours for each driver.
    ii. Prioritize drivers with fewer rides/hours when assigning new requests.
    iii. Consider factors like driver location and passenger destination to maintain efficiency
d. **Fairness Aspect**: Load balancing ensures a more equitable distribution of rides among drivers, promoting fairness in earnings. However, this may result in slightly longer wait times for passengers if the nearest driver has already completed many trips.

B2

Objective: Enhancing Rider Profit (D2)

1. **Efficient Route Planning**
   a. **Real-time Traffic Data Integration:** Adjust the weight of each edge in your graph to reflect current traffic conditions
      i. Add a function to fetch and process real-time traffic data
      ii. Update graph construction to include a parameter for traffic data which adjusts the weights of the edges
      iii. Ensure a_star_nx_optimized utilizes the updated graph for route calculations

```python
def fetch_real_time_traffic_data():
    # Fetch traffic data from an external API
    # This data should include information about traffic conditions on various road segments
    traffic_data = external_traffic_api.get_data()
    return traffic_data

def update_graph_with_traffic_data(graph, traffic_data):
    for edge in graph.edges():
        start, end = edge
        if (start, end) in traffic_data:
            # Update edge weight based on current traffic conditions
            graph[start][end]['weight'] = calculate_travel_time(traffic_data[(start, end)])
    return graph
```

Fig 1. Pseudocode Implementation

B2

Objective: Enhancing Rider Profit (D2)

2. **Dynamic Earnings Opportunities**
    a. **Dynamic Fare Calculations:** Implement a function within pair_drivers_passengers to calculate dynamic fares
        i. Utilize a fare calculation algorithm that adjusts prices based on factors like distance, current demand in the area (number of waiting passengers), and traffic conditions
        ii. Include this fare calculation in the ride pairing process, ensuring that drivers are aware of potential earnings from a ride before accepting

```
def calculate_dynamic_fare(distance, demand_factor, traffic_factor):
    base_fare = 2.0  # Base fare for a ride
    # Adjust the fare based on distance, demand, and traffic
    fare = base_fare + (distance * per_mile_rate) + (demand_factor * surge_rate) + (traffic_factor * traffic_rate)
    return fare

def pair_drivers_passengers_with_dynamic_fare(drivers, passengers, graph):
    # Existing pairing logic...
    for passenger in passengers:
        # Calculate dynamic fare for each potential ride
        fare = calculate_dynamic_fare(distance, demand_factor, traffic_factor)
        # Pair the driver and passenger considering the calculated fare
```

Fig 2. Pseudocode Implementation

B2

Objective: Enhancing Rider Profit (D2)

3. **Reduced Idle Time:**
    a. **Predictive Demand Modeling:** Enhance pair_drivers_passengers to suggest high-demand areas to idle drivers based on historical data and possibly other variables like time of day, weather, and local events
        i. Implement or integrate a machine learning model for demand forecasting
        ii. Use the model's output to guide idle drivers towards areas with expected high demand
        iii. Regularly update the model with new data to maintain accuracy

```
def predict_high_demand_areas(model, current_time):
    # Use the predictive model to forecast high-demand areas
    high_demand_areas = model.predict_demand(current_time)
    return high_demand_areas

def guide_idle_drivers(drivers, high_demand_areas):
    for driver in drivers:
        if driver_is_idle(driver):
            # Suggest high-demand areas to the driver
            driver['suggested_area'] = find_closest(high_demand_areas, driver['current_location'])

```

Fig 3. Pseudocode Implementation

B2

Objective: Ensuring Equitable Distribution of Rides

1. **Ride Allocation Fairness**
    a. **Equitable Ride Distribution:** Track each driver's completed trips or active hours and prioritize those with fewer rides or less active time
        i. Add tracking for each driver's ride count or active hours.
        ii. Use a priority system (such as a weighted queue) that considers this tracking in the ride assignment process
        iii. Ensure the system balances this fairness with geographical proximity to maintain efficiency for passengers

```
def update_driver_queue(driver_queue, drivers):
    for driver in drivers:
        # Update the queue based on the number of completed trips or active hours
        driver_queue.update(driver, driver['completed_trips'])

def fair_ride_assignment(driver_queue, passengers):
    for passenger in passengers:
        driver = driver_queue.pop_with_lowest_trips()
        pair_driver_with_passenger(driver, passenger)
```

Fig 4. Pseudocode Implementation

2. **Preference and Feedback Integration**
    a. **Driver Preferences:** Allow drivers to set preferences (e.g., preferred trip lengths, areas, times)
        i. Modify the drivers dataset structure to include preference fields.
        ii. Adjust the ride-pairing algorithm to consider these preferences in assigning rides.
        iii. Implement a feedback loop where drivers can update their preferences, and the system adapts accordingly

B2

Objective: Maintaining Good Performance for Passengers

1. **Regular Analysis and Adaptability**
    a. **Continuous Optimization and Feedback Analysis:** Establish a process for regularly reviewing the algorithm's performance in terms of driver satisfaction and earnings, as well as passenger wait times and service quality
        i. Set up metrics and KPIs (Key Performance Indicators) to measure the success of the algorithm from both the driver's and passenger's perspectives.
        ii. Regularly collect data on these metrics and use them to make informed adjustments to the algorithm.
        iii. Implement a feedback system for both drivers and passengers to report their experiences, feeding this data back into the system for continuous improvement
2. **Other Considerations**
    a. **Testing and Deployment:** Each of these modifications should be rigorously tested in a controlled environment before being deployed. This might involve A/B testing or pilot programs.
    b. **Scalability and Performance:** Ensure that the additional complexity does not significantly impact the performance of the algorithm, especially in terms of response time and scalability.

B3

1. Tracking Driver Locations and Routes

Strategy: Keep track of the current locations and intended routes of all active drivers to understand the density of the drivers in different areas

Modification:

a. After assigning a driver to a passenger, record the driver's route
b. Update the driver's location in real-time or at regular intervals

Impact: By constantly updating the locations and routes of drivers, the system gains a real-time overview of driver distribution across the network to identify potential congestion hotspots. With this data, the system can make more informed decisions about route planning and ride assignments, helping to distribute traffic more evenly and reduce the likelihood of contributing to congestion.

```
def update_driver_route(driver, route):
    driver['current_route'] = route
    # Update the driver's location periodically based on the route
    update_driver_location_periodically(driver)

def update_driver_location_periodically(driver):
    # Code to update driver's location over time
    pass
```

Fig 5. Pseudocode Implementation

B3

2. Congestion Prediction Model

Strategy: Implement a model that predicts congestion based on the number of drivers on specific routes using historical traffic data, current driver locations, and route choices

Modification:

a. Implement a congestion prediction algorithm.
b. Regularly feed data into this model to get real-time or near-real-time congestion predictions

Impact: A congestion prediction model that uses current driver locations and historical traffic data allows the system to forecast areas that are at risk of becoming congested. This proactive approach enables the system to adjust before congestion becomes problematic.

```
def predict_congestion(drivers):
    congestion_map = {}
    for driver in drivers:
        for segment in driver['current_route']:
            congestion_map[segment] = congestion_map.get(segment, 0) + 1
    return congestion_map

# Periodically update the congestion map based on real-time data
def update_congestion_map_periodically(drivers):
    # Periodic update logic
```

Fig 6. Pseudocode Implementation

Note: Regularly call update function to keep the congestion map updated with the latest data

B3

3. Dynamic Route Adjustment

Strategy: Use the congestion data to dynamically adjust routes for new ride assignments, avoiding areas predicted to be highly congested

Modification:

a. Integrate congestion data into the compute_path_length function.
b. Adjust the route choices based on congestion predictions

Impact: Adjusting routes dynamically in response to predicted congestion ensures that drivers are not all sent down the same paths. By considering real-time traffic conditions, including those caused by your drivers, the system can reroute drivers to less congested alternatives.

```python
# Similar to compute_path_length but adjusts weights based on congestion_map
def compute_path_length_with_congestion(graph, start_node, end_node, congestion_map):
    priority_queue = [(0, start_node)]
    distances = {node: float('inf') for node in graph}
    distances[start_node] = 0

    while priority_queue:
        current_distance, current_node = heappop(priority_queue)

        if current_node == end_node:
            return distances[end_node]

        for neighbor, weight in graph[current_node].items():
            # Adjust weight based on congestion
            adjusted_weight = weight * congestion_map.get((current_node, neighbor), 1)
            distance = current_distance + adjusted_weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heappush(priority_queue, (distance, neighbor))

    return float('inf')
```

Fig 7. Pseudocode Implementation

B3

4. Even Driver Distribution

Strategy: Modify the ride-pairing algorithm to avoid clustering too many drivers in the same area.

Modification:

a. In the pair_drivers_passengers function, consider the current distribution of drivers when assigning rides.

b. Prioritize pairing drivers who are in less congested areas, if feasible.

Impact: Modifying the ride-pairing algorithm to prioritize drivers in less congested areas helps prevent further exacerbation of traffic in already busy areas. By avoiding the assignment of too many drivers to the same area, the system not only reduces the likelihood of creating traffic bottlenecks but also potentially opens up more earning opportunities for drivers in less busy areas.

```python
def pair_drivers_passengers_evenly(drivers, passengers, graph, node_data, congestion_map):
    # Modified version of pair_drivers_passengers
    bst = build_binary_search_tree(node_data)
    driver_queue = [HeapElement(driver['Date/Time'], driver['ID'], driver) for driver in drivers]
    heapq.heapify(driver_queue)

    sorted_passengers = sorted(passengers, key=lambda p: p['Date/Time'])

    pairs = []
    while driver_queue and sorted_passengers:
        next_driver_element = heapq.heappop(driver_queue)
        next_driver = next_driver_element.data
        driver_location_idx = closest_node_bst(next_driver['lat'], next_driver['lon'], bst)

        # Find the best passenger considering congestion
        best_passenger = find_best_passenger(driver_location_idx, sorted_passengers, graph, congestion_map, bst)

        if best_passenger:
            # Update pairs, driver's route, and re-insert driver into the queue
            update_driver_route(next_driver, best_passenger['route'])
            updated_driver_element = HeapElement(next_driver['Date/Time'], next_driver['ID'], next_driver)
            heapq.heappush(driver_queue, updated_driver_element)
            pairs.append((next_driver, best_passenger))

    return pairs

def find_best_passenger(driver_location_idx, passengers, graph, congestion_map, bst):
    # Logic to find the best passenger considering congestion and other factors
    pass
```

Fig 8. Pseudocode Implementation

B3

Modification Overall Analysis:

1. **Efficiency and Scalability:** The proposed modifications are designed to be integrated into the existing system, enhancing its capability to handle a **larger volume of drivers without a significant increase in computational complexity**.
2. **Adaptability:** The system becomes more **adaptable to real-time conditions**, capable of making quick adjustments based on live data. This adaptability is key in a dynamic environment like urban traffic.
3. **Driver and Passenger Experience:** While focusing on reducing congestion, these modifications also aim to **maintain or improve the overall experience** for both drivers and passengers. Efficient routing reduces travel time and potentially increases the number of rides a driver can complete, while passengers benefit from reduced waiting times and quicker journeys.
4. **Broader Impact:** By addressing the issue of congestion proactively, the service not only improves its own operational efficiency but also contributes to the broader goal of **reducing urban traffic congestion**, which is a significant concern in many cities.

B4

1. Demand Prediction and Surge Pricing

Strategy: Utilize historical ride data to predict demand patterns in various locations and at different times. This information is crucial for implementing dynamic or surge pricing, where prices are adjusted based on predicted demand.

Application: Historical data about ride frequency, peak hours, special events, and even weather conditions are analyzed to forecast demand. This forecasting helps in setting surge pricing, ensuring a balance between supply and demand.

Implementation Details:

- Integrate a model that uses historical data to predict demand in different areas
- Based on the predicted demand, adjust pricing dynamically

```
def predict_demand(location, time, historical_data_model):
    predicted_demand = historical_data_model.predict(location, time)
    return predicted_demand

def calculate_dynamic_pricing(predicted_demand):
    if predicted_demand > demand_threshold:
        return base_rate * surge_multiplier
    return base_rate

# Modify the pair_drivers_passengers function to use dynamic pricing
def pair_drivers_passengers_modified(...):
    # ... existing logic ...
    for passenger in passengers:
        predicted_demand = predict_demand(passenger['location'], passenger['time'], historical_demand_model)
        dynamic_price = calculate_dynamic_pricing(predicted_demand)
        # ... remaining logic ...
```

Fig 9. Pseudocode Implementation

B4

2. Driver Position Optimization

Strategy: Analyze historical data to identify high-demand areas and suggest optimal positions for drivers to wait for ride requests

Application: Data from previous rides, including times and locations where demand was high, is used to create heat maps or recommendations for drivers, guiding them to areas where they are most likely to get ride requests.

Implementation Details:

- Use historical ride data to suggest optimal locations for drivers
- Update the driver recommendation system based on this analysis

```
def suggest_optimal_driver_positions(current_time, historical_data):
    optimal_locations = get_optimal_locations_from_historical_data(current_time, historical_data)
    return optimal_locations

# In the main function or a scheduled task
optimal_positions = suggest_optimal_driver_positions(current_datetime, historical_ride_data)
# Use these positions to guide drivers
```

Fig 10. Pseudocode Implementation

B4

3. Route Optimization Based on Past Traffic Data

Strategy: Use historical traffic data and past trip routes to optimize navigation and provide the fastest or most efficient routes

Application: By analyzing past trip data and traffic patterns, the algorithms can suggest routes that avoid common traffic bottlenecks, even predicting changes in traffic conditions based on time of day or day of the week.

Implementation Details:

- Analyze historical traffic data to optimize suggested routes
- Modify the route calculation function to consider this data

```
def compute_optimized_route(start_node, end_node, graph, historical_traffic_data):
    # Modify the route calculation logic to consider historical traffic patterns
    optimized_route = get_optimized_route_based_on_historical_data(start_node, end_node, graph, historical_traffic_data)
    return optimized_route
```

Fig 11. Pseudocode Implementation

B4

4. Estimated Time of Arrival (ETA) Accuracy

Strategy: Improve the accuracy of ETA predictions for passengers using historical ride data

Application: Historical data on route durations, traffic conditions, and even driver behavior patterns (like average speed) are used to provide more accurate ETA predictions for customers

Implementation Details:

- Use historical data to improve ETA predictions
- Integrate this data into the function that calculates ETAs

```
def calculate_accurate_eta(start_node, end_node, graph, historical_eta_data):
    # Use historical data to provide a more accurate ETA
    accurate_eta = get_eta_based_on_historical_data(start_node, end_node,
        graph, historical_eta_data)
    return accurate_eta
```

Fig 12. Pseudocode Implementation

B4

5. Improved Driver-Customer Matching

Strategy: Use historical ride data to improve the algorithm for matching passengers with drivers.

Application: Data about past rides, including wait times, cancellation rates, and passenger preferences, are used to enhance the customer-driver matching process, leading to reduced wait times and lower cancellation rates.

Implementation Details:

- Analyze past ride data to enhance the matching algorithm
- Factor in historical wait times, cancellation rates, and passenger preferences

```
def enhanced_customer_matching(driver, passenger, historical_matching_data):
    # Modify the matching logic based on historical data
    match_score = calculate_match_score(driver, passenger, historical_matching_data)
    return match_score

# Modify pair_drivers_passengers to use enhanced_customer_matching

```

Fig 13. Pseudocode Implementation