CS 581: RIDESHARING PROJECT REPORT

Shubadra Govindan, Amlaan Bhoi, Mohit Adwani, Debojit Kaushik
Department of Computer Science
University of Illinois at Chicago
{sgovin9, abhoi3, madwan2, dkaush4}@uic.edu

1 Introduction

1.1 Problem Statement

There is huge traffic congestion in the major cities all over the United States. On an average, 128 hours of driving time was lost per driver to congestion in 2018 in Los Angeles, California [1]. One of the factors responsible for the heavy congestion is increased car ownership. LA had an estimated 7.8 million vehicle registration in 2016 alone [2] while vehicle ownership is 1.62 vehicles per household [3].Apart from time, another crucial commodity is lost due to congestion which is fuel. Due to time and fuel wasted in traffic congestion, a typical driver loses over $2800 per year [4]. More fuel consumption releases more carbon monoxide and carbon dioxide in the air which leads to increase in air pollution and thus increase in health problems. One of the ways that the city is trying to tackle congestion is by planned expansion of the Los Angeles Metro Rail System [2]. Private companies like Uber and Lyft are trying to solve this problem by offering ride sharing services to decrease the number of cars on the road. Ride-sharing plays a crucial role in reducing congestion in 'hot spots' such as airports, major stations, and stadiums. In such places, a steady stream of passengers arrive via some public transport mode, say train, and then depart to different destinations. The large volume and continuous flow of arriving passengers create a golden opportunity for ride-sharing among departing passengers.

1.2 Objective

Develop a ride-sharing system to enable real-time taxi ride-sharing between customers(users) to optimize and reduce fuel consumption, and miles travelled, enabling users to save money and ease up traffic congestion. The user will input the destination address, number of travellers in party and delay/walking constraints. The aim is to devise a ride-sharing algorithm, to merge trips, enabling the users to share the rides given the constraints.

Following are the assumptions/constraints we adopt in this project:

- At most 2 passengers pooled together
- No walking by passengers
- Taxis are always available
- Static matching
- Source: *JFK*, Destination: *Manhattan*
- Delay time allowed for each passenger is 20% of original trip time

1.3 Previous Work

Uber Technologies Inc. offers peer-to-peer ride-sharing service called UberPool which matches riders heading in the same direction so that users can share the ride and cost. Uber restricts users to have only 2 passengers per request [5]. It does not provide a feature for the users to enter a delay or a walking constraint. Lyft Inc. also offers similar peer-to-peer ride-sharing service called Lyft Shared. Lyft, like Uber has the same restriction and does not provide a feature for the users to enter delay or walking constraint [6].

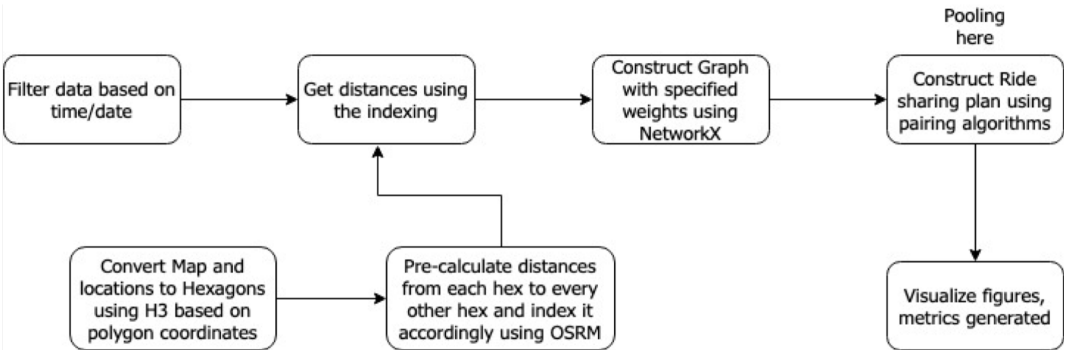

Figure 1: Overview of Process Flow

2 Data

2.1 NYC Taxi Cab Data

To simulate real time ride requests for the project we use the NYC Yellow Taxi Trip records from January 2016 to June 2016. This data has been provided by the NYC Taxi and Limousine Commission(TLC) [7]. Some of the features the data includes are pickup datetime, drop-off datetime, passenger count, trip distance, pickup location ID, drop-off location ID, etc. Due to the small scope of the project, we only consider rides which are originating from J.F. Kennedy Airport and terminating in Manhattan.

2.2 Maps and Locations

We use the Open Street Map data for this project. Only the New York state map data, which has been provided by GEOFABRIK, is used for finding the routes in the project since our rides are limited to JFK and Manhattan. The New York State map data (.osm.pbf file) is loaded onto the OSRM-Backed server which we will talk about in 2.4

2.3 Uber's H3 (Map Indexing)

H3 is a geospatial indexing system using a hexagonal grid that can be subdivided into finer and finer hexagonal grids. H3 is developed by Uber and is used in this project to represent the JFK airport and Manhattan Map in a hexagonal grid of 8 and 10 resolutions respectively. The polygon coordinates of JFK and Manhattan regions are passed in as a parameter in GeoJSON format to the H3 polyfill function to generate the hexagonal grids. The area of a single hexagon of the JFK airport map is approximately 0.73 km^2 and that of Manhattan airport map is approximately 0.015 km^2. Every ride request's pickup location and drop-off location is mapped to nearest hexagon using the H3 library. We precompute the shortest path distance and time for all possible pairs of the hexagons of JFK and Manhattan using the OSRM engine which we talk about in the next section 2.4. H3 library is also used to check whether the ride request is originating from JFK and terminating in Manhattan or not. We do this by mapping the pickup location ID (lat, long) and drop-off location ID (lat, long) to h3 hexagons and check if those hexagons are present in the source and destination for JFK and Manhattan respectively.

2.4 OSRM

Open Source Routing Machine (OSRM) is a high performance routing engine written in C++, designed to run on Open Street Map data. In this project, OSRM is used to precompute the shortest route distance and time for all possible pairs of the hexagons of JFK and Manhattan. Each hexagon is represented as a pair of latitude longitude values which is the center of that hexagon. The OSRM engine runs on Open Street Map data, which in this project is the New York state Map data obtained from Geofabrik.

3 Methodology

The entire pipeline and process can be found in Figure 1.

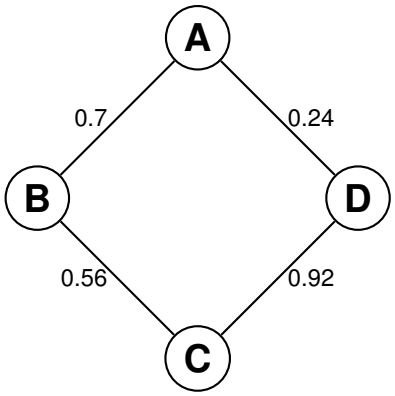

Figure 2: RSG with four trips and edges representing potential ride shares

3.1 Ride Sharing Graph

After finalizing on our data and how to represent physical locations in our logical representation, we now move on to how we consider rides and potential shares. We use the concept of **ride sharing graphs** (RSG). In a RSG, our nodes are individual trips (trips that are not yet shared but have individual trip details) while our edges are potential shares. The graph is represented using the `networkx` library in Python. In later sections, we define how we compute the edge weights for our RSG.

3.2 Precomputation

Using OSRM, we can determine the distance and duration taken for a taxi ride given hexagons H_1 and H_2. Given a hexagon H, we compute the distance and duration between said hexagon and every other hexagon H_i and store it in CSV files. We then combine all these values and store it in our final CSV database. We then create custom `Pandas` indexes from H_i, H_j pairs for faster indexing in run time.

3.3 Optimizations

Our main focus was speed and how we can increase efficiency of our computations. To aid this notion, we run all our experiments on a cluster of CPUs. We use 32 Intel Xeon CPUs in parallel. As we mentioned in the previous section, we create custom indices for our dataframe for faster indexing. This provides us a hashmap comparable lookup time in our dataframe. Speaking in terms of numbers, we have 3074 destination hexagons and 12 source hexagons. In total, we have $12 \times 12 + 12 \times 3704 + 3704 \times 3704 = 13764208$ rows. Our lookup operation now takes less than 1.2 milliseconds on average.

After indexing, our next bottleneck was edge weight calculations. Given a set of rides R within a pool time window (2, 5, or 10 minutes), we have S rides for each pool group. We thus need to compute edge weights for all rides r_i within S. This usually took 6-7 minutes when we considered one day's rides. Thus, to optimize this, we use Python's `multiprocessing` package to parallelize our edge weight calculations. After parallelizing our code, the calculations now take 50 seconds to 1 minute.

3.4 Merging Conditions/Assumptions/Constraints (Edge Weight Calculation)

Merging conditions refer to the specific variables that determine if two rides can be pooled together or merged together into one ride. In this experiment, we employ multiple factors that determine if two rides can be merged. In theory, some possible factors can be distance, time for each passenger, time for the whole shared ride, the amount of delay each passenger is willing to bear, social factors to better match passengers, and more.

Before we go over the specific factors we consider in this project, it is vital to define the trip sequences we consider. Following the template defined by Santi et. al. [8], given two individual trips a and b, we consider the following possible shared trip sequences:

- $D_{p(a)\to p(b)} + D_{p(b)\to d(a)} + D_{d(a)\to d(b)}$
- $D_{p(a)\to p(b)} + D_{p(b)\to d(b)} + D_{d(b)\to d(a)}$

- $D_{p(b)\to p(a)} + D_{p(a)\to d(a)} + D_{d(a)\to d(b)}$
- $D_{p(b)\to p(a)} + D_{p(a)\to d(b)} + D_{d(b)\to d(a)}$

These four are the possible sequences that we can pool two rides together. In this project, we consider the following factors to determine if two trips can be merged:

1. **Distance saved for merged trip:** The overall trip distance saved for the merged trip
2. **Time constraint *for each* passenger:** Depending on the sequence of pickup and drop-off, the time delay each passenger is willing to take for a shared ride vs individual rides
3. **Social score affinity between passengers:** The social affinity between two passengers given their profession and language

Distance We consider the distances between each hexagon's central point as our distance from any point x to y. As mentioned, these distances are pre-computed and stored in-memory for fast lookup. Distance comparisons are easy as we sum up all the possible distances from each of our sequence and find the minimum one that satisfies the following conditions:

$$\begin{aligned} D &\leq D_A, D_B \\ D &\leq D_{min} \end{aligned} \tag{1}$$

where D is the possible shared distance in consideration, D_a and D_b are individual distances of trips A and B respectively, and D_{min} is the current minimum distance encountered.

Time It is obvious that sharing rides may increase the total ride time of each passenger involved when compared to individual ride times. The question becomes how we can minimize this extra time with respect to the delay the passengers are willing to take. In our formulation, we consider the following cases for incorporating time in our calculations:

$$\begin{aligned} T_{s_A} &\leq T_A \\ T_{s_B} &\leq T_B \\ T &\leq T_{min} \end{aligned} \tag{2}$$

where T_{s_A}, T_{s_B} are new trip times for passenger A, B respectively, T_A, T_B are original trip times for A, B and T_{min} is the current minimum time encountered.

One thing to notice is that T_{s_A} and T_{s_B} are calculated based on the proposed sequence. For example, if A is picked up first, then B is picked up, then A is dropped off, then B is dropped off, the time for A is only till he gets dropped off. B's time is the whole trip as he waits for the taxi to come to him and then eventually gets dropped off. These times are variable depending on the order of pickup and dropoff based on the four possibilities listed above.

Social Affinity We also consider social affinity between passenger A and B. We consider two social factors: *profession* and *language*. The range of professions we consider are: musician, engineer, doctor, lawyer, and actor. The range of languages we consider are: English, French, Spanish, and Hindi. We then create a symmetric affinity matrix that gives a score to the corresponding professions and languages. All the scores are within the range of 0 and 1.0.

Once we have the calculations for the distances, times, and social scores, we also assign three weights w_1, w_2, w_3 to each of our factors. following is how we calculate the final edge weights in our ride-sharing graph:

$$\begin{aligned} dist &= w_1 \times (D_A + D_B - D_{min}) \\ time &= w_2 \times (T_A + T_B - T_{min}) \\ social &= w_3 \times (\frac{PF + LA}{2}) \\ E &= social \times (dist + time) \end{aligned} \tag{3}$$

where E is the final edge weight assignment for trips A and B. The pseudocode for the edge weight calculation algorithm is found in Algorithm 1.

3.5 Ride Matching Algorithm

Once our ride sharing graph (RSG) has been constructed with appropriate edge weights, we can now construct matching sets. Our matching sets contain nodes of individual trips and edges with maximum weights that minimize distance,

Algorithm 1 EWC algorithm

procedure GENERATE(a, b) ▷ Rides a and b
    $\mathbf{R}$ = Generate all possible ride combinations with distances and durations
    return $\mathbf{R}$
procedure EWC(a, b) ▷ Rides a and b
    tmin, dmin $= \infty$
    $\mathbf{R}$ = {**generate**(a, b)} ▷ Generate all ride combinations with a, b
    for each t, d in $\mathbf{R}$ **do**
        if $t < (T_a + \delta(a))$ and $t < (T_b + \delta(b))$ and $t <$ tmin and $d < (d_a + d_b)$ and $d \leq$ dmin **then**
            tmin $= t$
            dmin $= d$
    S = Calculate social score
    return $S \times ((D_a + D_b - \text{dmin}) + (T_a + T_b - \text{tmin}))$

time, and maximize social score. We use the **Maximum Weighted Matching** algorithm which is also known as the **Blossom's Algorithm** [9]. The Blossom Algorithm is a means of finding the maximum matching in a graph, through the Blossom contraction process. This polynomial time algorithm is used in several applications including the assignment problem, the marriage problem, and the Hamiltonian cycle and path problems (i.e., Traveling Salesman Problem). The algorithm takes $\mathcal{O}(n^3)$ time complexity.

We will not go over the exact algorithm as it is quite verbose to explain and detail here. Instead, we shall go over some key concepts used in the algorithm. The blossom algorithm takes a general graph $G = (V, E)$ and finds a maximum matching M. The algorithm starts with an empty matching and then iteratively improves it by adding edges, one at a time, to build augmenting paths in the matching M. Adding to an augmenting path can grow a matching since every other edge in an augmenting path is an edge in the matching; as more edges are added to the augmenting path, more matching edges are discovered. The blossom algorithm has three possible results after each iteration. Either the algorithm finds no augmenting paths, in which case it has found a maximum matching; an augmenting path can be found in order to improve the outcome; or a blossom can be found in order to manipulate it and discover a new augmenting path.

3.5.1 Augmenting Paths

An augmenting path is an odd length path that has unmatched nodes for endpoints. The edges in an augmenting path alternate: one segment will be in the matching, the next segment is not in the matching, the next segment is in the matching, and so on. The algorithm works by finding augmenting paths so it can increase the size of the matching by one each iteration. To do this, it starts by finding unmatched vertices (vertices that are not connected to an edge that is in the matching). Then, it builds up an alternating path from these nodes. This means that the path begins at a node not in the matching — to be an augmenting path, it must start and end with unmatched nodes.

The algorithm continues to build on the augmenting path until it can't anymore. When it stops building an augmenting path, it is either the case that the graph has run out of edges that are not already in the augmenting path, in which case, there is no maximum matching, or the algorithm gets to a point where an augmenting path cannot be made, at which point the algorithm returns a maximum matching.

3.5.2 Blossoms

The idea behind blossoms is that an odd-length cycle in the graph can be contracted to a single vertex so that the search can continue iteratively in the now contracted graph. Here, these odd-length cycles are the blossoms. The algorithm can just go on through the graph and treat the tricky cycle as if it were only a single node — fooling the algorithm into thinking there is an even number of nodes in the graph (it has explored so far) so that it can run the Hungarian algorithm on it.

A blossom is a cycle in G consisting of $2k + 1$ edges of which exactly k belong to M, and where one of the vertices in the cycle (the base) is such that there exists an alternating path of an even length, called a stem, from to an exposed vertex w. The advantage is given due to the ease at which the stem part can be toggled between edges in the blossom. The goal of the algorithm is to shrinks a blossom into a single node in order to reveal augmenting paths. It works by running the Hungarian algorithm until it runs into a blossom, which it then shrinks down into a single vertex. Then, it begins the Hungarian algorithm again. If another blossom is found, it shrinks the blossom and starts the **Hungarian algorithm**, and so on.

3.6 Software and Technologies Used

The project primarily uses Python 3. Below are a few Python libraries used in the project -

- `requests`
- `pandas`
- `numpy`
- `h3` (Uber H3)
- `matplotlib`
- `networkx`
- `json`
- `datetime`
- `multiprocessing`

External modules used -

- Open Source Routing Machine
- Jupyter Notebook
- Collaboration Platform: Github

4 Results

4.1 Evaluation Metrics

The following metrics have been considered to evaluate our system.

- **Pool Window Time:** A variable pool window time has been used to analyze the performance of the algorithm. Pool windows of 2, 5 and 10 minutes have been evaluated.
- **Utilization:** We define utilization as the number of rides that were pooled divided by the number of original rides.

$$Utilization = \frac{rides_{pooled}}{rides_{original}} \tag{4}$$

- **Distance Saved (as a percentage):** This metric gives us the distance saved by combining the trips as compared to the original individual trip distance.

$$\%distancesaved = \frac{distance_{original} - distance_{pooled}}{distance_{original}} * 100 \tag{5}$$

- **Trips Saved (as a percentage):** We define trips saved as the number of trips pooled together divided by the original number of trips. This value is multiplied by hundred and is represented in the form of a percentage.
- **Computational Time:** The time taken by the system to process and generate pools for the given pool window time. As mentioned above, the code is executed on a cluster of CPUs.

4.2 Output Plots

The different output plots generated for this project cover different metrics - such as percentage utilization (as defined above), distances saved, trips saved, computation time, and effect of social score. The data tested for the algorithm ranges from January to June 2016. We have performed day-wise comparisons for June 2016 to understand the impact of days of the week on the pooling strategy. A month-wise comparison of the metrics mentioned has also been visualized. Observations and key takeaways are discussed in the following section. Figures 3 to 9 showcase our results.

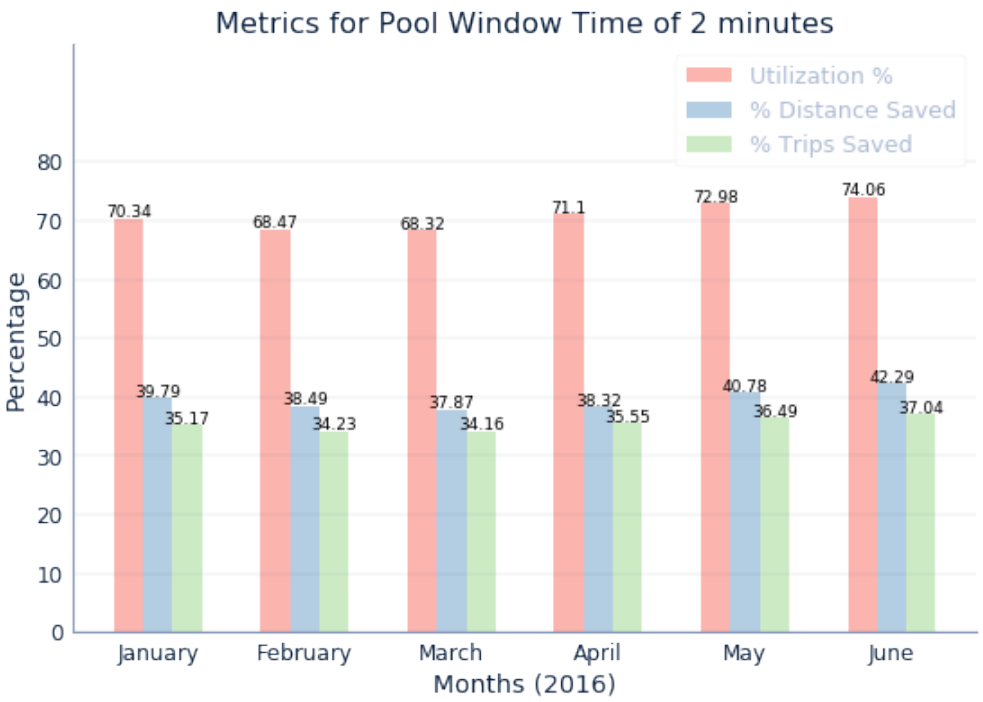

Figure 3: Metrics for pool window time 2 minutes

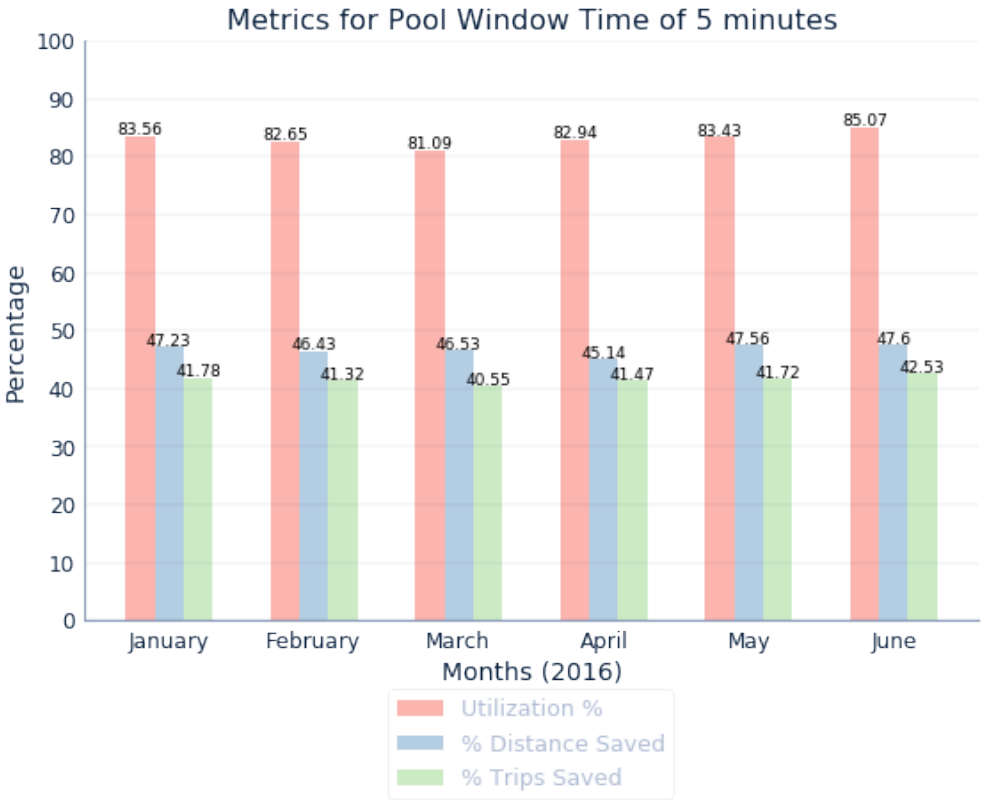

Figure 4: Metrics for pool window time 5 minutes

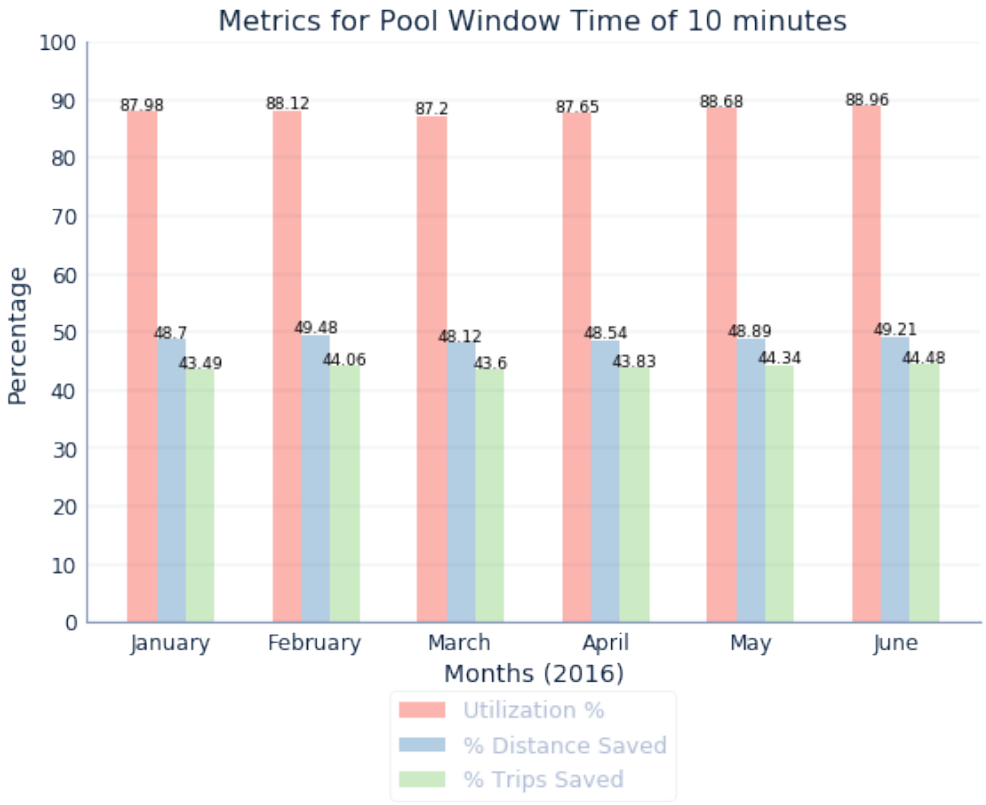

Figure 5: Metrics for pool window time 10 minutes

4.3 Observations and Trends

One of our primary objectives was to vary the pool time window and study its effects. We selected the windows as 2, 5 and 10 minutes. Further, we ran our algorithm on six months of data - from January to June 2016. A few interesting observations -

- Larger the pool window time, higher the utilization percentage
- Varying the pool window time between 2 minutes and 5 minutes showed a considerable difference in the distances saved: the lowest percentage distance saved for 2 minutes was around 37%, while for 5 minutes it was around 45%
- June had the most favorable results (that is, higher utilization, distance saved and trips saved). March scored low on all metrics - for varying pool windows
- On performing a day-wise comparison (on June 2016 data), we noticed that Fridays and Mondays had the highest number of shared rides, with pool window times as 5 and 10 minutes. This could be explained by increase in number of people flying in to JFK for the new week / weekend
- Saturdays and Sundays in June 2016 had the lowest utilization
- Computation time was positively correlated with the pool window time - higher the pool window time, larger the computation time
- Factoring social score did not improve the performance. We have intuitively assigned social scores based on languages and professions. Due to this, there is not much of a difference between the ride sharing algorithms with and without social score

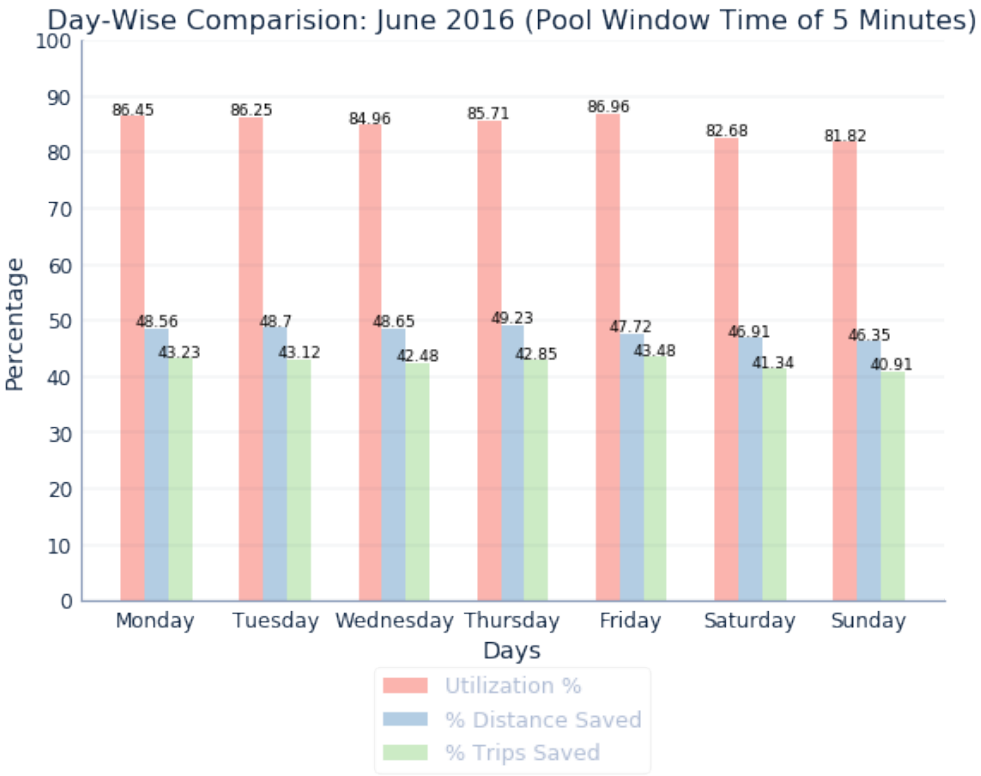

Figure 6: Metrics for day-wise comparison for pool window time 5 minutes

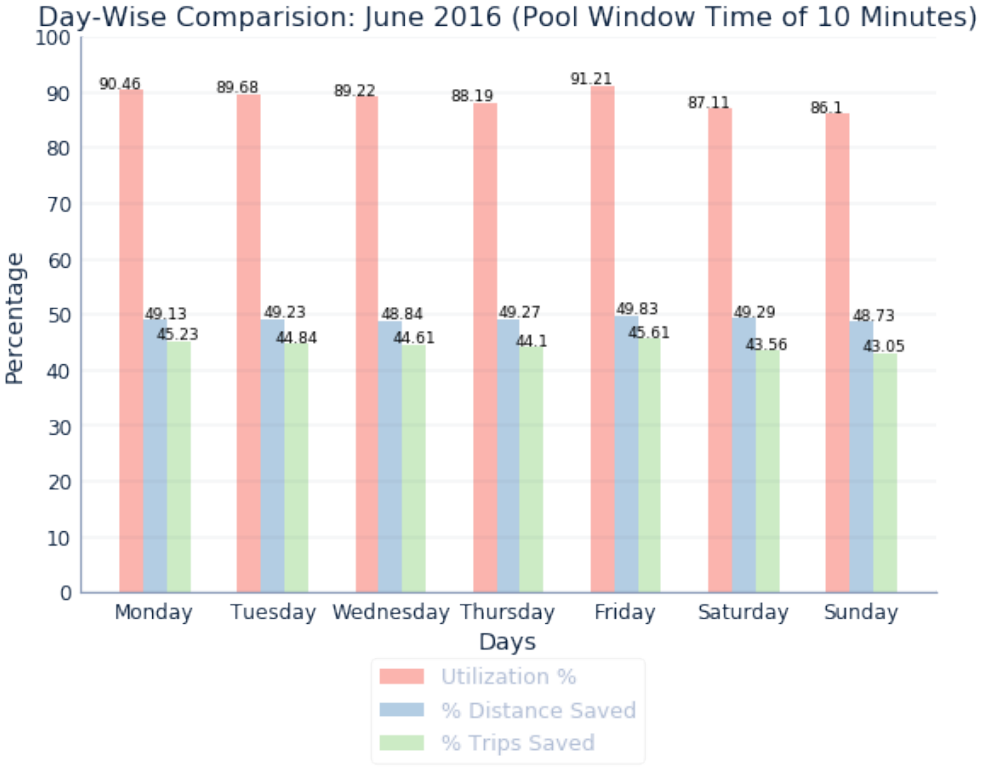

Figure 7: Metrics for day-wise comparison for pool window time 10 minutes

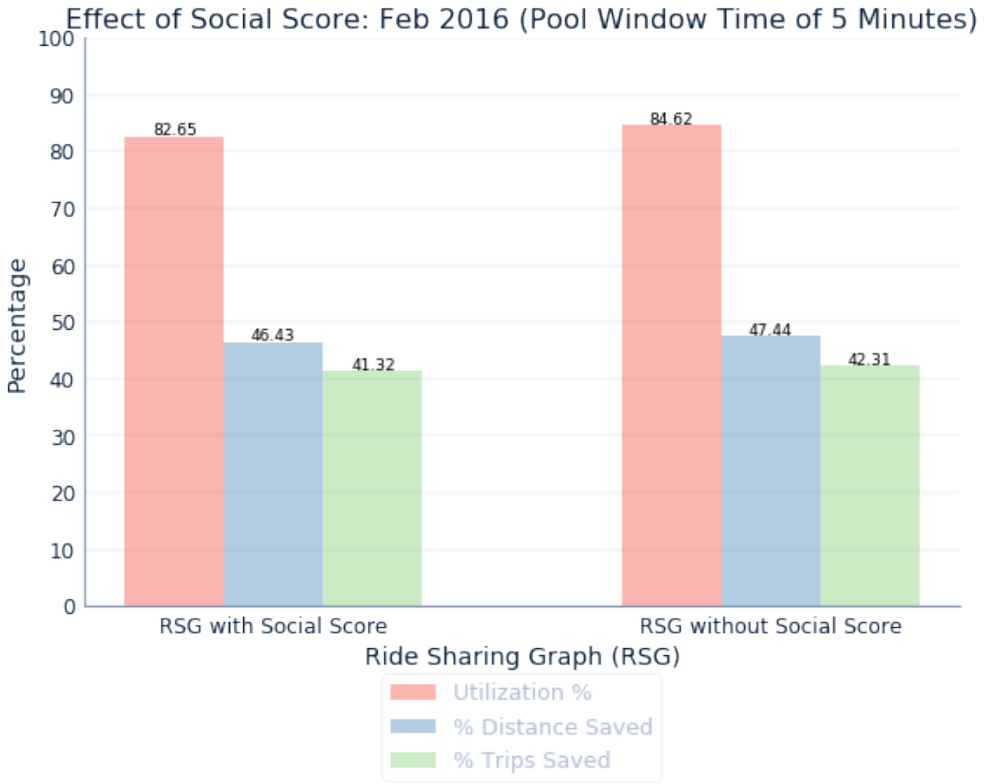

Figure 8: Effect of Social Score in our Ride Sharing Algorithm

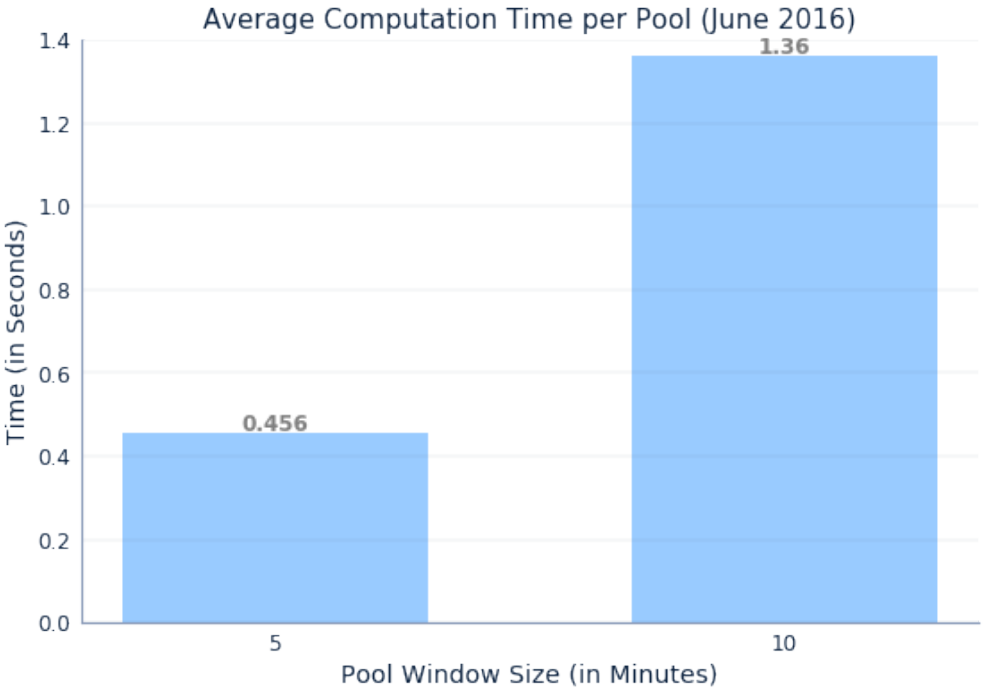

Figure 9: Average Computation Time for Variable Pool Windows - June 2016 Data

5 Conclusion

5.1 Summary

In conclusion, we created and simulated a ride pooling system to study the performance of our graph algorithms and their trade-offs. First, we sourced the data from NYC Taxi Data set for the months of January to June 2016. We created custom Python scripts to scrape the CSV files from the website. Next, we used Uber's H3 hexagon tiling system for segregating our map. We then obtained polygon boundary coordinates for JFK and Manhattan. Feeding these polygon coordinates to H3 got us the relevant hexagon pickup and drop off points. Then, we snapped our rides pickup and drop off at the closest hexagon center. We then used OSRM's API to procure distances and duration as necessary. Next, we generated the ride sharing graph using NetworkX library. We then used our custom logic for edge weight calculations. Finally, we used NetworkX's `max_weight_matching` algorithm for generating maximum matching sets. Varying pool window times showed the effectiveness of our ride sharing. It highlighted how the number of rides considered positively correlates with the utilization percentage. With this project, we were able to discover nuances of ride sharing and gather more insight into metrics for better performance and optimization.

5.2 Future work

For future work, we can incorporate the following considerations into our system:

- Consider fare estimation as a criteria for matching rides
- Consider road architecture (one-way, blocked, etc)
- Consider dynamic matching/routing
- Consider more than k=2 people per ride
- Allow passengers to change destination dynamically
- Consider density estimation for surge pricing

References

[1] Niall McCarthy. Infographic: The u.s. cities with the worst traffic problems, Feb 2019.

[2] Andrea Lo. Los angeles' traffic problem in graphics, Feb 2018.

[3] Governing. Vehicle ownership in u.s. cities data and map.

[4] Baruch Feigenbaum and Rebeca Castaneda. Los angeles has the world's worst traffic congestion - again, Apr 2018.

[5] What is uberpool | what is carpool.

[6] Lyft Inc. About shared rides.

[7] TLC. Tlc trip record data, 2018.

[8] Paolo Santi, Giovanni Resta, Michael Szell, Stanislav Sobolevsky, Steven H Strogatz, and Carlo Ratti. Quantifying the benefits of vehicle pooling with shareability networks. *Proceedings of the National Academy of Sciences*, 111(37):13290–13294, 2014.

[9] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys (CSUR)*, 18(1):23–38, 1986.