Taxi Ride Sharing Analysis

CS480 Fall 2020 Final Project Report

University of Illinois at Chicago

Group Members

- Angela Timochina - atimoc2@uic.edu
- Reem Hussein - rhusse3@uic.edu
- Eldin Vujic - evujic2@uic.edu

Project code can be found on Github, but screenshots of queries included in the report for clarity.

Table of Contents

Data Window Used

For this project, we originally wanted to analyze 12 months of data from both Green and Yellow taxis in the New York Taxi Data. However, given the magnitude of this data, our machines were not powerful enough to process that much data. Instead, we utilized four months of data. We used the following New York Taxi Data:

- Green Taxi January 2014
- Green Taxi February 2014
- Yellow Taxi January 2014
- Yellow Taxi February 2014

In doing so, we were able to add, process, clean, and apply our algorithm on a large amount of data while also saving time and machine power.

Data Cleaning

The first thing we did was add a ride ID to the data table to act as the primary key for our main algorithm. We did this using the following query:

Query 1: Primary Key ride_id Addition

```
ALTER TABLE green_1_january ADD ride_id INT AUTO_INCREMENT PRIMARY KEY;
ALTER TABLE green_2_february ADD ride_id INT AUTO_INCREMENT PRIMARY KEY;

ALTER TABLE january_green ADD ride_id INT AUTO_INCREMENT PRIMARY KEY;
ALTER TABLE february_green ADD ride_id INT AUTO_INCREMENT PRIMARY KEY;
```

As previously stated, we needed this ride_id in order to access elements later when we apply our main algorithm. In addition, we created a cleaning script that would remove all the noise from the data and remove all the observations that are not complete. We also separated the single and double riders into two different datasets so that we can use that in our querying strategy. It also determined the speed of the ride using the following function written here.

This query is used in our main clean up query, where we extract viable rides and to grab single and double riders. We used the same query over 4 months of taxi data for green and yellow taxi data, which would result in 8 different datasets to separate our single and double riders. Even though this may seem a lot, we combine the data with another query later. This query is exhibited below, but can also be accessed here.

Query 2: Single and Double Rider Clean Up on Green Taxi Data

```
# get the single riders
CREATE TABLE jan_green_one_rider
SELECT ride_id, pickup_datetime, dropoff_datetime, pickup_longitude, pickup_latitude,
dropoff_longitude, dropoff_latitude, passenger_count, trip_distance,
get_speed(pickup_datetime, dropoff_datetime, trip_distance) AS 'speed'
FROM rideshare.january_green
WHERE pickup_latitude != 0 AND pickup_longitude != 0 AND dropoff_latitude != 0 AND
dropoff_longitude != 0 AND passenger_count = 1 AND trip_distance > 0 AND
pickup_datetime != dropoff_datetime
ORDER BY pickup_datetime, pickup_latitude, pickup_longitude;

# get the double riders
CREATE TABLE jan_green_two_rider
SELECT ride_id, pickup_datetime, dropoff_datetime, pickup_longitude, pickup_latitude,
dropoff_longitude, dropoff_latitude, passenger_count, trip_distance,
get_speed(pickup_datetime, dropoff_datetime, trip_distance) AS 'speed'
FROM rideshare.january_green
WHERE pickup_latitude != 0 AND pickup_longitude != 0 AND dropoff_latitude != 0 AND
dropoff_longitude != 0 AND passenger_count = 2
AND trip_distance > 0 AND pickup_datetime != dropoff_datetime
ORDER BY pickup_datetime, pickup_latitude, pickup_longitude;

CREATE TABLE feb_green_one_rider
SELECT ride_id, pickup_datetime, dropoff_datetime, pickup_longitude, pickup_latitude,
dropoff_longitude, dropoff_latitude, passenger_count, trip_distance
FROM rideshare.february_green
WHERE pickup_latitude != 0 AND pickup_longitude != 0 AND dropoff_latitude != 0 AND
dropoff_longitude != 0 AND passenger_count = 1
AND trip_distance > 0 AND pickup_datetime != dropoff_datetime;

CREATE TABLE feb_green_two_rider
SELECT ride_id, pickup_datetime, dropoff_datetime, pickup_longitude, pickup_latitude,
dropoff_longitude, dropoff_latitude, passenger_count, trip_distance
FROM rideshare.february_green
WHERE pickup_latitude != 0 AND pickup_longitude != 0 AND dropoff_latitude != 0 AND
dropoff_longitude != 0 AND passenger_count = 2
AND trip_distance > 0 AND pickup_datetime != dropoff_datetime;
```

Query 3: Single and Double Rider Clean Up on Yellow Taxi Data

```
# get the single riders
CREATE TABLE jan_yellow_one_rider
SELECT ride_id, pickup_datetime, dropoff_datetime, pickup_longitude, pickup_latitude,
dropoff_longitude, dropoff_latitude, passenger_count, trip_distance,
get_speed(pickup_datetime, dropoff_datetime, trip_distance) AS 'speed'
FROM rideshare.january_yellow
WHERE pickup_latitude != 0 AND pickup_longitude != 0 AND dropoff_latitude != 0 AND
dropoff_longitude != 0 AND passenger_count = 1 AND trip_distance > 0 AND
pickup_datetime != dropoff_datetime
ORDER BY pickup_datetime, pickup_latitude, pickup_longitude;

# get the double riders
CREATE TABLE jan_yellow_two_rider
SELECT ride_id, pickup_datetime, dropoff_datetime, pickup_longitude, pickup_latitude,
dropoff_longitude, dropoff_latitude, passenger_count, trip_distance,
get_speed(pickup_datetime, dropoff_datetime, trip_distance) AS 'speed'
FROM rideshare.january_yellow
WHERE pickup_latitude != 0 AND pickup_longitude != 0 AND dropoff_latitude != 0 AND
dropoff_longitude != 0 AND passenger_count = 2
AND trip_distance > 0 AND pickup_datetime != dropoff_datetime
ORDER BY pickup_datetime, pickup_latitude, pickup_longitude;

CREATE TABLE feb_yellow_one_rider
SELECT ride_id, pickup_datetime, dropoff_datetime, pickup_longitude, pickup_latitude,
dropoff_longitude, dropoff_latitude, passenger_count, trip_distance
FROM rideshare.february_yellow
WHERE pickup_latitude != 0 AND pickup_longitude != 0 AND dropoff_latitude != 0 AND
dropoff_longitude != 0 AND passenger_count = 1
AND trip_distance > 0 AND pickup_datetime != dropoff_datetime;

CREATE TABLE feb_yellow_two_rider
SELECT ride_id, pickup_datetime, dropoff_datetime, pickup_longitude, pickup_latitude,
dropoff_longitude, dropoff_latitude, passenger_count, trip_distance
FROM rideshare.february_yellow
WHERE pickup_latitude != 0 AND pickup_longitude != 0 AND dropoff_latitude != 0 AND
dropoff_longitude != 0 AND passenger_count = 2
AND trip_distance > 0 AND pickup_datetime != dropoff_datetime;
```

In doing this, we result in 8 new datasets, 2 for each of the months we are analyzing. While we like having everything separated, we did merge our single riders to make a more efficient database. We also took the advice given to us in our final presentation where we merge datasets from green and yellow taxis together in order to maximize the number of combinable trips. For example, we combined the single riders for both green and yellow taxi data from the month of january, turning two datasets into one and allowing us to find more combinable trips. We used the following query to achieve this.

Query 4: Combining Green and Yellow Datasets

```
# all january single riders from green and yellow taxis
CREATE TABLE jan_single_riders
SELECT * FROM rideshare.jan_green_one_rider
UNION
SELECT * FROM rideshare.jan_yellow_one_rider;

# all january double riders from green and yellow taxis
CREATE TABLE jan_double_riders
SELECT * FROM rideshare.jan_green_two_rider
UNION
SELECT * FROM rideshare.jan_yellow_two_rider;

# all february single riders from green and yellow taxis
CREATE TABLE feb_single_riders
SELECT * FROM rideshare.feb_green_one_rider
UNION
SELECT * FROM rideshare.feb_yellow_one_rider;

# all february double riders from green and yellow taxis
CREATE TABLE feb_double_riders
SELECT * FROM rideshare.feb_green_two_rider
UNION
SELECT * FROM rideshare.feb_yellow_two_rider;
```

In doing this, we halved the amount of datasets we need to search on and are able to maximize the number of trips to combine when running our main algorithm queries. We then made sure to grab the total trip distance of all our datasets prior to running our algorithm to effectively compare and contrast our data.

Figure 1. Total Miles Before Running Algorithm

Month	Total Trip Distance (Miles)
January Single Riders	28,087,889.78
January Double Riders	5,703,226.03
February Single Riders	27,228,455.13
February Double Riders	5,386,885.69

Main Algorithm Database Querying

The first step was to clean the tables as outlined above. Once we were able to get the tables into the correct format with the addition of speed calculation we then utilized the split single and double riders to try and optimize not only distance saved but the amount of riders as well, i.e. max number is 3 passengers. By utilizing two separate tables it was easier for us to pair two rides based on certain conditions such as taxi being able to pick-up/ drop-off within certain delay times. This information was populated into a new data table with other information such as the distance between the pickup points in the combined trip, the distances of all middle sequences of the trip, the dropoff distance for the combined rides, the total distance of the trips prior to their combinations, and the average speed of the combined trips.

This first query extracted the rides combinable and calculated the distance for all sections of a trip and possible center sequences. We noticed that regardless of the taxi order pickup or dropoff the distance between those two points will be the same. Thus, the only part of the combined trip that will have varying distances is the center part. Therefore, if we minimize the center part of the trip, the overall trip will be the optimized distance. This optimized distance is then later used to determine which trips to merge based on the maximized savings.

Query 5: Finding all Combinable Rides with Delay as ride_one_combo

```
CREATE TABLE ride_one_combo
SELECT r1.ride_id as ride_one_id, r2.ride_id as ride_two_id,
    TIMESTAMPDIFF(SECOND, r1.pickup_datetime , r2.pickup_datetime) AS time_diff,
    (r1.speed+r2.speed)/2 as speed,
    r1.trip_distance + r2.trip_distance as original_distance,
    get_distance_mi (r1.pickup_latitude, r1.pickup_longitude, r2.pickup_latitude, r2.pickup_longitude) AS distance_pickup,
    get_distance_mi (r1.pickup_latitude, r1.pickup_longitude, r1.dropoff_latitude, r1.dropoff_longitude) AS middle_o1d1,
    get_distance_mi (r1.pickup_latitude, r1.pickup_longitude, r2.dropoff_latitude, r2.dropoff_longitude) AS middle_o1d2,
    get_distance_mi (r2.pickup_latitude, r2.pickup_longitude, r1.dropoff_latitude, r1.dropoff_longitude) AS middle_o2d1,
    get_distance_mi (r2.pickup_latitude, r2.pickup_longitude, r2.dropoff_latitude, r2.dropoff_longitude) AS middle_o2d2,
    get_distance_mi (r1.dropoff_latitude, r1.dropoff_longitude, r2.dropoff_latitude, r2.dropoff_longitude) AS distance_dropoff
FROM jan_green_two_rider r2, ride_one r1
WHERE TIMESTAMPDIFF(SECOND, r1.pickup_datetime , r2.pickup_datetime) > 0 AND TIMESTAMPDIFF(SECOND, r1.pickup_datetime , r2.pickup_datetime) < 5*60;
```

A trip can consist of 4 possible middle sequences o1d1, o1d2, o2d1, and o2d2 where o is origin, d is destination, and 1,2 represents the ride. The distances for these middle sections was calculated through the function get_distance_mi. In addition, this query found the average speed of the possible combined trips.

The next query took the above table defined as ride_one_combo and found which distance is the minimum and populated a new table based on if there was a distance savings with the combined new total distance. The minimum was found through the use of the function get_min_middles.The query is shown below.

Query 6: Filtering ride_one_combo by distance savings > 0 as ride_combo

```
CREATE TABLE ride_combo
SELECT ride_one_id, ride_two_id, speed,
       distance_pickup + distance_dropoff + get_min_middle(middle_o1d1, middle_o1d2, middle_o2d1, middle_o2d2) as total_distance,
       original_distance - (distance_pickup + distance_dropoff + get_min_middle(middle_o1d1, middle_o1d2, middle_o2d1, middle_o2d2)) as distance_savings,
       (original_distance - (distance_pickup + distance_dropoff + get_min_middle(middle_o1d1, middle_o1d2, middle_o2d1, middle_o2d2)))/original_distance*100 as percent_savings
from ride_one_combo
WHERE original_distance - (distance_pickup + distance_dropoff + get_min_middle(middle_o1d1, middle_o1d2, middle_o2d1, middle_o2d2)) > 0;
```

The resulting table, ride_combo, may have repeated ride_id's and repeated rides being used in a combination which is why the next query strategy looked for the trips where the distance savings was maximized on the first ride_id. Then this strategy is repeated on the second ride_id column and the resulting final table has a unique combination of rides based on the maximized distance saved. The maximization queries are shown below.

Query 7: Maximizing Distance Savings and Getting Unique ride_combo ride_id's

```
create table ride_combo_two
select distinct *
from ride_combo
GROUP BY ride_one_id
HAVING percent_savings = max(percent_savings)
order by ride_one_id, percent_savings DESC;

select distinct *
from ride_combo_two
GROUP BY ride_two_id
HAVING percent_savings = max(percent_savings);
```

The final result is a table with all the distinct ride combinations labeled with the two ride_ids, the trip speed, the total_distance of the combined trip, the distance_savings calculated from the original_distance of the original two individual trips subtracting the total_distance, and the percent_savings which is the amount of distance saved over the original_distance. All this information can then be used to determine the average percent savings of the entire algorithm.

Functions

One of the things we implemented in our project that assisted with the implementation and flow of our algorithm was the incorporation of three functions. These three functions were the get_distance_mi, get_min_middle, and get_speed functions.These functions helped speed up the runtime of our algorithm which notably made the algorithm more efficient. These functions

also enabled us to do multiple calculations throughout the algorithm without having to reuse the same code. Ultimately these functions helped our algorithm have a peak in performance by being able to calculate multiple values for the purpose of saving miles and combining compatible taxi rides.

Function 1. get_distance_mi

```
DROP FUNCTION get_distance_mi;
DELIMITER //

CREATE FUNCTION get_distance_mi(origin_lat DOUBLE, origin_lon DOUBLE, dest_lat DOUBLE, dest_lon DOUBLE) RETURNS DOUBLE DETERMINISTIC
BEGIN
 RETURN (( 3959 * acos( cos( radians(origin_lat) ) * cos( radians(dest_lat) )
   * cos( radians(origin_lon) - radians(dest_lon)) + sin(radians(dest_lat))
   * sin( radians(origin_lat))));
END
//
DELIMITER ;
```

This function aided our algorithm by giving us the total distance between any two given cords which was crucial when the algorithm was dealing with multiple origins and destinations. This algorithm would take in the origins/destinations given by the latitude and longitude then calculate the total distance between the origins and the destinations. Lastly it would return the distance that was calculated between these two points.

Function 2. get_min_middle

```
DROP FUNCTION get_min_middle;
DELIMITER //
CREATE FUNCTION get_min_middle(m1 DOUBLE, m2 DOUBLE, m3 DOUBLE, m4 DOUBLE) RETURNS DOUBLE DETERMINISTIC
BEGIN
    IF m1 < m2 and m1 < m3 and m1 < m4 then
        return m1;
        end if;
    if m2 < m1 and m2 < m3 and m2 < m4 then
        return m2;
        end if;
    if m3 < m1 and m3 < m2 and m3 < m4 then
        return m3;
        end if;
    if m4 < m1 and m4 < m2 and m4 < m3 then
        return m4;
        end if;
    return 0;
END
//
DELIMITER ;
```

This function helped utilize the difference sequences that were introduced in the algorithm. Essentially what this function did was it helped calculate and find the minimized distance between any given trip while also giving us the maximized savings. This aided our algorithm by reducing the amount of inaccurate combined trips it may receive. To further explain this, it basically allowed us to check all the possible combinations between a trip to many other destinations which helped determine which trip pairs had the most efficient minimal distance traveled while at the same time maximizing the savings.

Function 3. get_speed

```
DROP FUNCTION get_speed;
DELIMITER //

CREATE FUNCTION get_speed(pu_date DATETIME, do_date DATETIME, distance DOUBLE) RETURNS DOUBLE DETERMINISTIC
BEGIN
    RETURN(distance/TIMESTAMPDIFF(SECOND,pu_date,do_date));
END
//
DELIMITER ;
```

This function aided with calculating the speed of any distance. This was important to our algorithm because it allowed us to get the speed of the current distance which was calculated by using the formula of speed, $S = D/T$, where $D = Distance$ and $T = DO\ date - PU\ date$, where the *PU date* = pick up location date time, and the *DO date* = drop off location date time. This bit of information allowed our algorithm to calculate the speed at any given location along with the correlating dropoff/pickup locations.

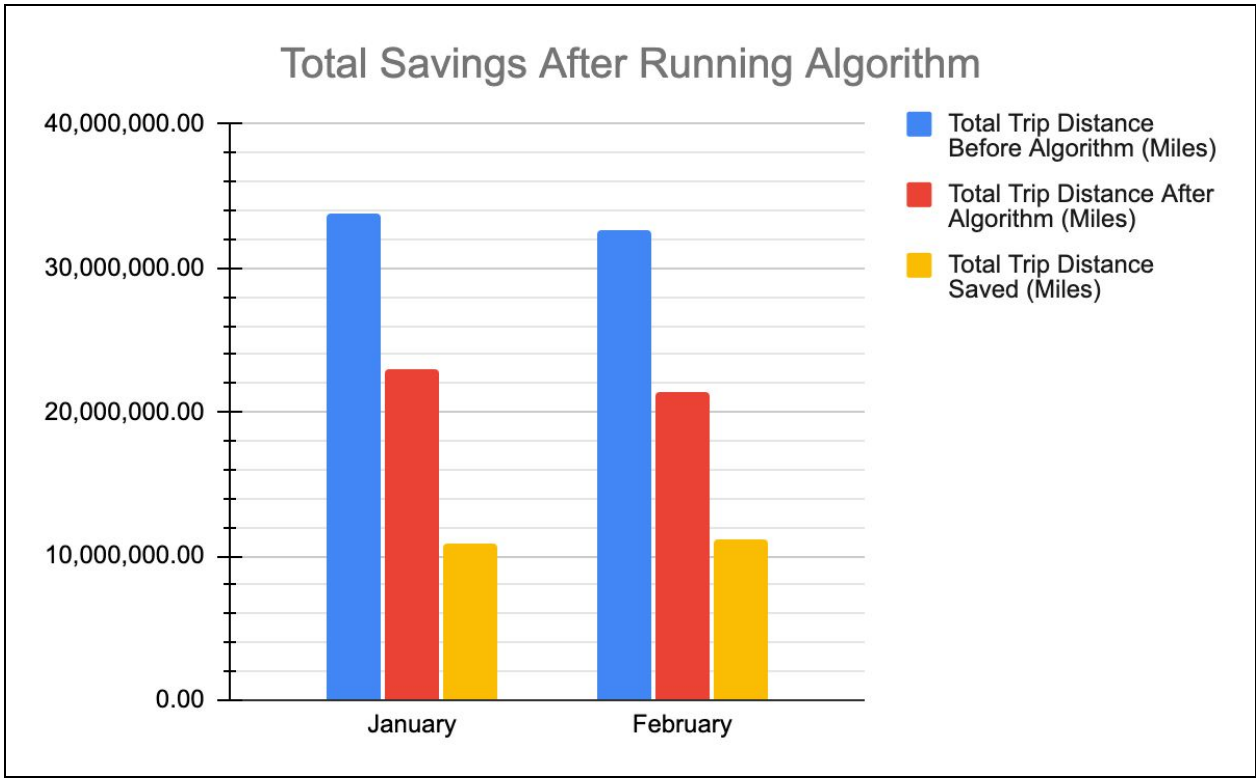
Effectiveness

To showcase the effectiveness of this strategy, grabbed the total distance saved percentage of that specific month. For each month, we included the total trip distance before our algorithm, total trip distance after our algorithm, the total distance saved as a result, and its percentage.

Figure 2. Total Savings After Running Algorithm

Month	Total Trip Distance Before Algorithm (Miles)	Total Trip Distance After Algorithm (Miles)	Total Trip Distance Saved (Miles)	Total Trip Distance Saved (Percentage)
January	33,791,115.81	22,970,186.79	10,820,929.02	32.023%
February	32,615,340.82	21,462,525.03	11,152,815.79	34.195%

Figure 3. Total Savings Visualization



As we can see, our algorithm did an impressive job at saving miles and combining trips once we combined our single riders from yellow and green taxi data. As mentioned in our final project presentation, prior to merging that data, we were saving an average of 28.07% miles across both taxis in January and February. Upon merging the data, we can see the average for both months increased, giving us an average of 33.109% savings for the entire project and a total of 21,973,744.81 miles saved across the months of January and February in 2014 for green and yellow taxi data.

Average Query Run Time

Clean Up Script

The total time it took to clean up four months worth of taxi data, from both yellow and green taxi data, took 412.526 seconds. The average time it took to make each of the temporary datasets was 51.566 seconds per query. This average was calculated by taking the total time and dividing it by the number of total queries (412.526 / 8). Even though some of these queries were significantly faster than others, some of the months had more data than others, which explains this discrepancy. This is why even though some tables only took less than 2 seconds, those that took almost 3 minutes averaged those out.

The total time it took to combine the datasets into four datasets, monthly by single and double riders, took 201.682 seconds. The average time it took to make each of these datasets was 50.421 seconds per query. This average was calculated by taking the total time and dividing it by the number of total queries (201.682 / 4).

Main Algorithm Script

When running the main algorithm script, we found that the time taken to run the script on a dataframe limited to 10000 rows was about 20 seconds. The total dataframe for the month of January is about 1100 times this size so it would take approximately 220000 seconds, or 6 hours, to run the entire algorithm on the January riders, taking a little more than 12 hours total to run this algorithm. This could have been smaller had we been more efficient with the way that we wrote our algorithm, however we wanted to be honest with the time it took to actually conduct this on our data. In addition, we did not have strong enough machines to run this algorithm faster, which could have also contributed to the high time it took to query.

Delay

We included a 5 minute delay on our algorithm which was applied to each ride in our data sets. This allowed us to analyze rides from both types of taxis, in addition to allowing us to track which combined rides were arriving within the 5 minute delay compared to those that were not able to carry out within this time limit.

Limitations and Issues Raised in Presentation

One of the things that arose during our final project presentation was us not merging the green and the yellow taxi data so that our datasets included single riders from both taxis for that month. Originally, we did not want to mix the data in an effort to not corrupt any of the data. After reflecting on this, we realized that we are not maximizing the benefits of our algorithm because we are not including all the possible mergeable rides together.

To rectify this, we decided that after separating the yellow and green taxi data into single and double riders, we would combine the single riders from each type of taxi from the same month so that we can find more trips to merge with the algorithm. This can be explained more in this section of our final report.

Teamwork retrospective analysis

Peer Evaluations

Originally, we had 4 members in our group. However, due to their inactivity, they asked us to remove their name off the project. With regards to that, we all had to work a little harder to make

sure our project would be ready for the end of the semester. We have each rated each other below so that we all submit a peer evaluation of each other. In doing so, we were able to make sure everyone's point of view was acknowledged and can see how each person's contribution was perceived by others in the group. So, we each gave a **rating out of 5** in how we think we each did and how each person worked on this project using a self and peer evaluation. We will also list everyone's individual contributions.

Reem Hussein

Criteria	Reem	Angela	Eldin
Contributed to the success of mid semester presentation	4.5	3	1.5
Contributed to the success of final semester presentation	5	5	2
Attended group meetings regularly	4.5	4.5	3
Contributed meaningfully to group discussions	5	5	2.5
Prepared work in a quality manner	5	5	2

Angela Timochina

Criteria	Reem	Angela	Eldin
Contributed to the success of mid semester presentation	4.75	3.5	2
Contributed to the success of final semester presentation	5	5	3
Attended group meetings regularly	4.5	4	4
Contributed meaningfully to group discussions	5	5	3
Prepared work in a quality manner	5	4.75	2

Eldin Vujic

Criteria	Reem	Angela	Eldin
Contributed to the success of mid semester presentation	4	3	2
Contributed to the success of final semester presentation	4.5	5	2.5
Attended group meetings regularly	4.5	4	4
Contributed meaningfully to group discussions	5	5	4
Prepared work in a quality manner	5	4.25	2

Contributions of Individual Group members

Reem Hussein

- Implement and edit the cleaning script with Angy
- Did finalizations to all scripts for increased readability
- Main contributor to the final project presentation
- Main contributor to the final project report

Angela Timochina

- Main contributor to the cleaning script
- Main contributor to the final algorithm
- Contributed to helper functions used in the final algorithm
- Explained algorithm in final project report

Eldin Vujic

- Helped implement the main algorithm / prototype algorithm
- Helped with presenting the presentations
- Helped fix the prototype algorithm, making it have a minimum search value

Appendix

Our full SQL queries can be found on github. In order to run this program successfully, please follow the instructions below.

- Download MySQL for your operating system using this link.
- Download and install MySQL Workbench.
- Download the green and yellow taxi data from January and February of 2014 from TLC Trip Record Data.
- Create your data tables and load your tables using the respective scripts found here.
- Run the cleaning script on the downloaded data, making sure to pay attention to the naming conventions you have given for your datasets. We recommend using those mentioned in the script to avoid altering the queries.
- Load out helper functions into your rideshare schema. The functions can be found here.
- Run the full ride share algorithm, which can be found here making sure to pay attention to the following
    - The current script only runs the algorithm on the January data. We adjusted the name of the month for this algorithm, so when ready to run it on February data, make sure to adjust the naming convention in the algorithm.
    - This takes a long time to run, so adjust your MySQL settings so that MySQL does not lose connection during the query running.