CS 581 - Database Management Systems

University of Illinois at Chicago

(Spring 2020)

RideSharing Project Report

<u>Team - 06</u>

Apurva Raghunath

Ashwin Balasubramani

Nandana S Prasad

Siddhanth Venkateshwaran

Contents

1. Introduction & Objective

1.1 Background

This project evaluates ride-sharing algorithms on spatio-temporal data. The data in this case represents nearly 700 million trips in New York City. While the emergence of novel Transportation Network Companies (e.g. Uber) has helped increase the supply of drivers during peak times, they have done little to reduce cost in 'hot spots' such as airports and major stations. The large volume and continuous flow of passengers create a golden opportunity for ridesharing. In a ride sharing mechanism, the passenger submits a ride request shortly before departure specifying some parameters such as pickup time, passenger destination, willingness to walk to the destination, the maximum delay tolerated due to ride-sharing, maximum time the passenger is willing to walk, and number of travelers in the party. In general, the goal is to allocate passengers to taxi cabs for ride sharing given various constraints.

1.2 Objective

Our objective is to maximize the distance saved using rideshare while keeping the passengers' walking and delay constraints in consideration.

2. Assumptions and Restrictions

To implement the system within the time frame and analyze the results we limited our approach by keeping certain assumptions as listed below:

- The number of trips that can be merged is capped to 2; this aligns with the fact that most of the vehicles can accommodate 2 passengers at max
- Social Preferences of passengers are not considered
- Trips originate and end at LaGuardia Airport
- Traffic conditions are disregarded
- The maximum de-tour time any passenger can bear is not more than 20% of total estimated travel time
- Tolerable walk time of any passenger is assumed to be 10 minutes at most.
- An average speed of 35 mph is assumed to compute travel times for some pair of points.
- Points within 2km from laguardia's exact coordinates are assumed to be laguardia.

Period of Time Evaluated : Whole month of January 2016, 1st week of February 2016 and 17 May 2016

3. Experiments Conducted (Dataset Filtration and Precomputation)

3.1 Technical Components

The entire project was developed using Python 3.7 on PyCharm IDE. The data was stored in the PostgreSQL database for Windows 10. Openrouteservice API was used to determine point-to-point travel distance and time, and appropriate dropoff locations for passengers.

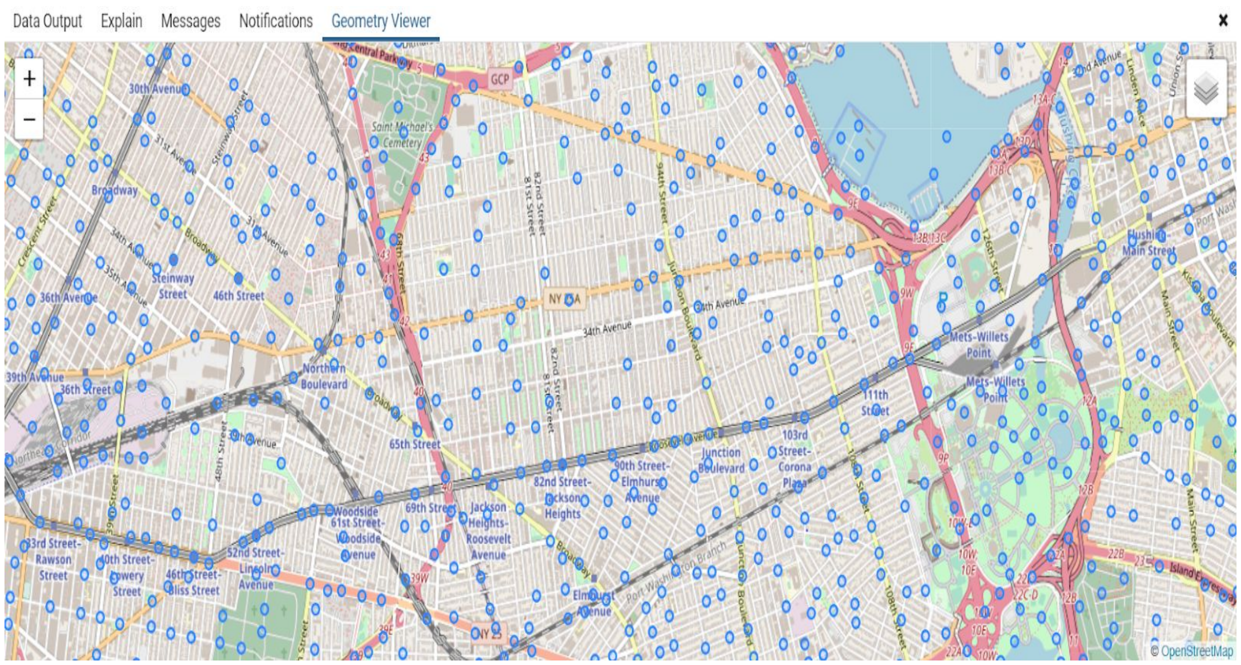
Before we move directly to algorithms we would like to explain the data filtration and pre computation performed.

3.2 Dataset Considered

The dataset was provided by the NYC Taxi and Limousine Commission. The official yellow taxi trip records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off coordinates, trip distance, itemized fares, rate types, payment types and driver-reported passenger count. However, the fields itemized fares, rate types, payment types and passenger count were not considered for the implementation of the project.

3.3 Data Filtration

- Trips with Source as LaGuardia - All trips with pickup points farther than 2km from LaGuardia were removed.
- Trips with Destination as LaGuardia - All trips with destination points farther than 2km from LaGuardia were removed.
- Part of New York was exported from OpenStreetMaps as a Planet OSM file, containing over 5 million intersections. This file was loaded into a PostgreSQL database enabled with PostGIS, a spatial database extension.
- To reduce the search space and to reduce the size of the set of points to be considered for precomputation, a subset of the points was filtered out using a threshold distance of 100 metres, forming a grid-like structure.
- This final set consisted of 10796 points uniformly spaced out across Manhattan, Brooklyn, Queens, etc which are at least 100 metres apart from each other.
- A part of this final set of points is shown below (indicated by the blue points):


Data Output
Explain
Messages
Notifications
Geometry Viewer
© OpenStreetMap

3.4 Precomputation of drop-off points

Reachability is a crucial component for many businesses from all different kinds of domains. To this end, the OpenRouteService api was used to obtain isochrones which helped us determine which area's objects (intersections) are reachable within a given time or distance.

Working of Isochrones is described below:

The Isochrones Service of OpenRouteService API was used to precompute the drop off points for each point in our final set of points. To determine reachable areas from any given location(s) on the map, the isochrones service returns these regions as contours of polygons. Input can be of 2 types, namely time or distance as well as corresponding intervals. If the input entails a list of points then the service will return intersections of the computed isochrones if any exist.

Computing drop-offs using OpenRouteService Isochrones

In our project this isochrone service came handy for computing the set of reachable (drop off) points within a given distance or time for a particular point coordinate. This procedure was repeated for all the points contained in the final set. Also, for each point from the final set, only those drop offs were considered which were within the 10 minute walk time constraint. Lastly the final set of points along with the drop-offs were stored in a PostGreSQL database for querying.

3.5 Precomputation of Travel distance/time

Openrouteservice’s time-distance matrix service allows us to obtain time and distance information between a set of locations (origins and destinations) and returns them in a structured JSON response. This API is extremely convenient and scalable for batch requests determining aggregated metrics of routes. Similar to locations and directions, the transportation mode and compute routes which adhere to certain restrictions, such as avoiding specific road types or object characteristics may also be specified. One prominent use case which can be built on top of this service is to easily explore the fastest or shortest combination of a set of destinations which should be reached – this is commonly referred to as the traveling salesperson problem.

Computing distance and time using OpenRouteService Time-Distance Matrix

We utilized this distance matrix service for computing the matrix of travel distance and time units between LaGuardia and every other point under consideration. However, the OpenRouteService API has certain request restrictions. One can make upto 40 requests per minute and upto 500 requests per day. For each of these requests a maximum of 59 points can be provided to compute the distance and time between every pair of points in the input set in the form of a matrix.

To overcome this rate limit, we also employed PostGIS for the distance computation.

Distance and time computed using the API is stored in the PostGreSQL database for querying, instead of calling the api for every other trip.

PostGIS

PostGIS is a spatial database extension for PostgreSQL object-relational databases. It adds support for geographic objects allowing location queries to be run in SQL. Technically PostGIS was implemented as a PostgreSQL external extension. PostGIS offers many features rarely found in other competing spatial databases such as Oracle Locator/Spatial and SQL Server. Some of these features are mentioned below:

- Geometry types for Points, LineStrings, Polygons, MultiPoints, MultiLineStrings, MultiPolygons and GeometryCollections.
- Spatial predicates for determining the interactions of geometries using the 3x3 DE-9IM (provided by the GEOS software library).
- Spatial operators for determining geospatial measurements like area, distance, length and perimeter.
- Spatial operators for determining geospatial set operations, like union, difference, symmetric difference and buffers (provided by GEOS) and many more.
- Spatial Indexes using R-tree structured bounding boxes.

PostGIS's Distance Sphere function was used to overcome the API's limitation. We use the PostGIS function ST_DistanceSphere for this purpose. This function returns the minimum distance in meters between two lon/lat points. It uses a spherical earth and radius derived from the spheroid defined by an SRID(Spatial Reference Identifier). However, after a number of trial and error, it was found that the function has an error of approximately 3000 meters in the distance returned compared to the distance obtained using the OpenRouteService API. Thus 3000m was added to the distance obtained from the ST_DistanceSphere in order to compensate for the error.

Spatial Indexes have been enabled for POINT objects in the table holding the travel distances/times, and the table holding the final set of 10796 discretized points for faster querying. The index is the GIST (Generic Index Structure) index which sets the index method to use the R-Tree instead of the default B-Tree, which maintains indexes on the entire geometry of an object, instead of just the box end coordinates like the R-Tree does. This makes the R-Tree index more lossy but faster to query the geometries.

3.6 Precomputation of Request Pools

All the trip requests obtained from the NYC yellow trip data is grouped together into different pools. We have considered four such pools namely, pool of 5 minutes from LaGuardia, pool of 10 minutes from LaGuardia, pool of 5 minutes to LaGuardia, pool of 10 minutes to LaGuardia. The procedure used to sort the trips into respective pools is as follows:

- Sort the trip request dataset for a particular day based on the pickup datetime column starting from midnight.
- Iterate through each trip checking the pickup time and noting the start time for each of the pools and add those trips to the respective pools. For example, say that the first trip started at 12:00 am on a particular day. So all the trips requested until the first 5 minutes will go into the 5 min pool and the requests from 12:06am onwards will be outside this first pool. If this request is for source as LaGuardia, then the trip goes to the pool, 5 min from LaGuardia.
- Repeat the procedure for all the trips of that particular day.

Given below is an example of pre-computation of pools applied on the NYC yellow trip dataset for May 17th, 2016.
Total number of trips from LaGuardia - 12796
Total number of trips to LaGuardia - 5792
After applying the above procedure we get the following information:
For trips from LaGuardia:

- 5 min pool had a total of 211 pools with an average of 70 trips per pool
- 10 min pool had a total of 120 pools with average of 150 trips per pool

For trips to Laguardia:

- 5 min pool had a total of 222 pools with an average of 30 trips per pool
- 10 min pool had a total of 127 pools with an average of 70 trips per pool

Thus computed pools are fetched on the go and stored in a separate json file having the pickup list, destination list and the trip distance list for each of the pools.

The algorithm proposed iterates through each of these pools and each trip in the pool for further processing and to determine the shareability of the trips.

4. Algorithms Evaluated

Step 1: Specify the trip file and the trip direction to run the algorithm on.

Step 2: All the incoming trip requests are segregated into a pool size of 5 minutes and pool size of 10 minutes respectively.

Step 3: For the pool fetched, every trip combination is considered for shareability.

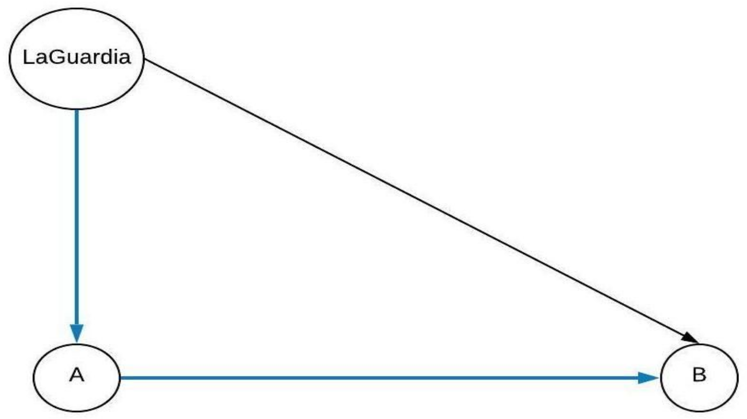
Step 4: Given two source or destination point coordinates, drop off points for both points are queried from the PostgreSQL database.

Step 5a: Shareability conditions are first evaluated when no-walking is involved. If the trips are sharable, these trips (along with the value of the distance saved) are inserted in a shareable list and the next pair is checked for shareability. Otherwise shareability conditions are evaluated when walking is involved.

Step 5b: If the trips end at LaGuardia, then the walking case is not considered.

Step 6: After every trip has been processed, the list of shareable trips is passed to the maximum weight matching algorithm.

Trips from LaGuardia (No Walking)



A - Destination of Passenger 1 B - Destination of Passenger 2

Case 1 (Passenger 1 is dropped first at A - indicated by the blue path):

1. **Dist**(LaG, A) + **Dist**(A, B) < **Dist**(LaG, A) + **Dist**(LaG, B)
2. **Time**(LaG, A) + **Time**(A, B) <= **Time**(LaG, B) + **Delay**(Passenger 2)

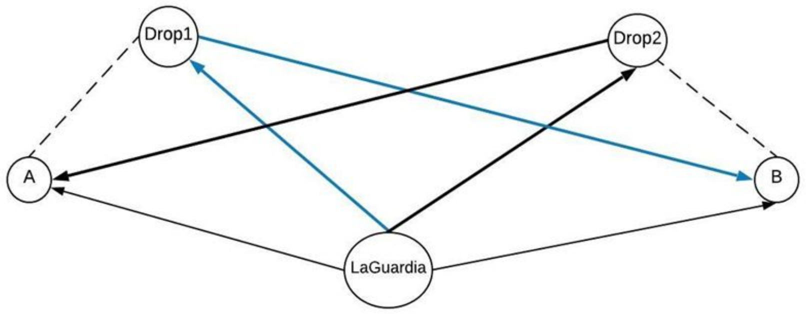
If both of these conditions are satisfied then the trips are shareable otherwise we consider Case 2.

Case 2 (Passenger 2 is dropped first at B):

1. **Dist**(LaG, B) + **Dist**(B, A) < **Dist**(LaG, A) + **Dist**(LaG, B)
2. **Time**(LaG, B) + **Time**(B, A) <= **Time**(LaG, A) + **Delay**(Passenger 1)

If both of these conditions are satisfied then the trips are shareable otherwise we move on to the next pair of trips.

Trips from LaGuardia (Walking Considered)



Consider the above figure. Passenger 1's destination is A and Passenger 2's destination is B. Suppose Drop1 and Drop 2 are the drop off points for Passenger 1 and Passenger 2 respectively. The trips are shareable if any one of the following conditions are satisfied:

Case 1 (Passenger1 is dropped first at Drop1 - indicated by the blue path) :

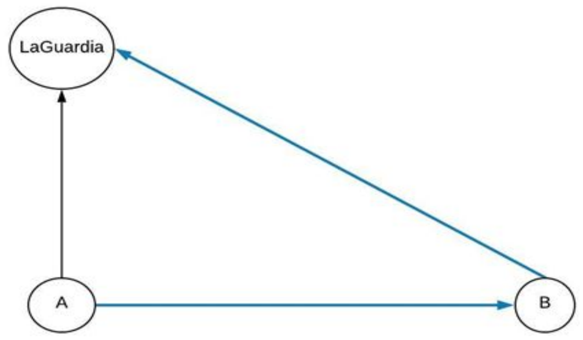
1. **Dist**(LaG, Drop1) + **Dist**(Drop1, B) < **Dist**(LaG, A) + **Dist**(LaG, B)
2. **Time**(LaG, Drop1) + **Time**(Drop1, B) <= **Time**(LaG, B) + **Delay**(Passenger 2)

The above conditions will be evaluated for every drop point associated with the destination of Passenger 1. If none are satisfied then we move on to the next case where Passenger 2 will be dropped first.

Case 2 (Passenger2 is dropped first at Drop2 - indicated by the black path)

3. **Dist**(LaG, Drop2) + **Dist**(Drop2, A) < **Dist**(LaG, A) + **Dist**(LaG, B)
4. **Time**(LaG, Drop2) + **Time**(Drop2, A) <= **Time**(LaG, A) + **Delay**(Passenger 1)

Trips to LaGuardia (Walking is not considered)



Suppose 2 passengers are starting from Source A and Source B respectively and the destination is LaGuardia in both cases. The trips are shareable if any one of the set of conditions below are satisfied:

Case 1: Pick up Passenger 1 at A, then pick up Passenger 2 at B and then go to LaGuardia(indicated by the blue path in the figure).

1. **Dist**(A , B) + **Dist**(B, LaG) < **Dist**(A, LaG) + **Dist**(B, LaG)
2. **Time**(A , B) + **Time**(B, LaG) < **Time**(A, LaG) + **Delay**(Passenger 1)

If both of these are satisfied then push both the trips in the shareable list and move on to the next pair of trips, otherwise consider Case 2.

Case 2: Pick up Passenger 2 at B, then pick up Passenger 1 at A and then go to LaGuardia.

1. **Dist**(B, A) + **Dist**(A, LaG) < **Dist**(A, LaG) + **Dist**(B, LaG)
2. **Time**(B, A) + **Time**(A, LaG) <= **Time**(B, LaG) + **Delay**(Passenger 2)

If both of these conditions are satisfied, then push the trips in the shareable trips and move on to the next pair. Otherwise, these trips are not shareable.

In every case where trips are shareable, the distance saved is calculated as follows:
Distance Saved = (Total Distance of Unshared Trips) - (Total Distance of Shared Trips)

For the previous case where passenger 2 was picked first and then passenger 1, the distance saved would be :
Distance Saved = (**Dist**(A, LaG) + **Dist**(B, LaG)) - (**Dist**(B, A) + **Dist**(A, LaG))

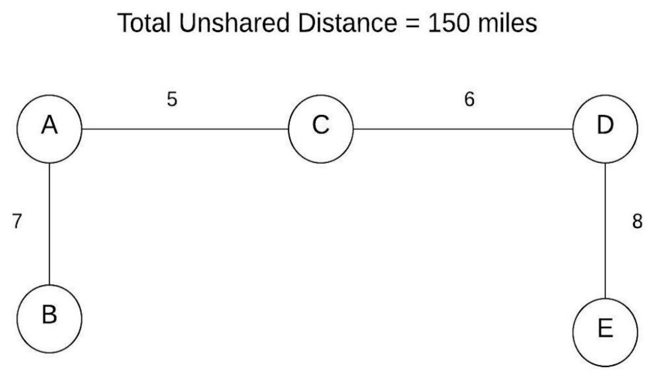
This value would be the value of the edge formed between the 2 shared trips in the final ride sharing graph.

4.1 Maximum Weight Matching

The maximum weight matching part of the algorithm is used as a Python library implemented by Joris van Rantwijk. The idea is based on the paper “Efficient Algorithms for Finding Maximum Matching in Graphs" by Zvi Galil, ACM Computing Surveys, 1986. The paper uses the algorithm invented by Jack Edmonds, which finds augmenting paths in a graph and uses the ‘primal-dual’ method to find a matching with the maximum weight.

The maximum weight matching library takes the input as a list of tuples, with each tuple representing an edge in the final ride-sharing graph formed after evaluating the shareability conditions. The trips are represented by non-negative integers and each tuple as (Trip1, Trip2, Edge Distance value).

For example:

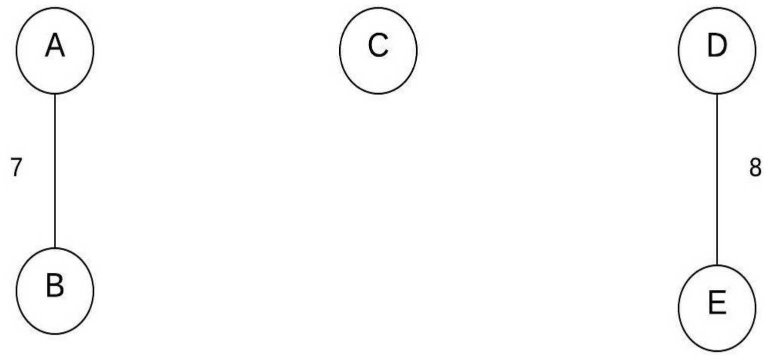


For the above ridesharing graph, we consider the trips as :

A - 1, B - 2, C - 3, D - 4, E - 5.

The input to the library would be : [(1, 2, 7), (1, 3, 5), (3, 4, 6), (4, 5, 8)]

The expected output for the above graph is:



The output of the library is a list where each index 'i' (starting from 1) of the list represents a trip and the value is :

a. -1, if the trip at index 'i' is not matched/shared with any other trip.

b. Index of the trip which is shared with the current trip at index 'i'.

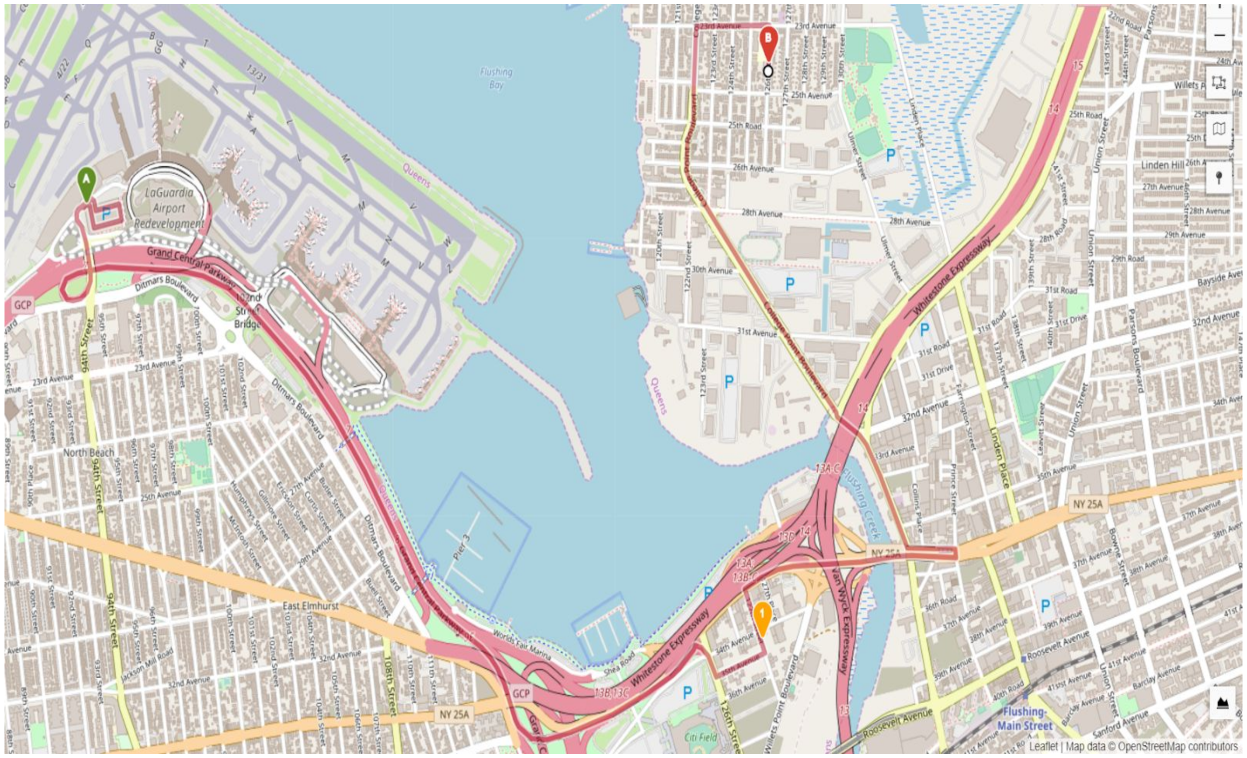
For the above graph the output would be: **[2, 1, -1, 5, 4]**

Note that the list starts at index 1. The trip C at index 3 is not shared with any other trip so it has -1 as its value.

5. Test Simulations

To test the shareability conditions evaluated, we took random destination points in New York, the source as LaGuardia Airport and implemented the above algorithm mentioned, without performing the maximum weight matching step. Walk time and delay time were considered to be 10 minutes.

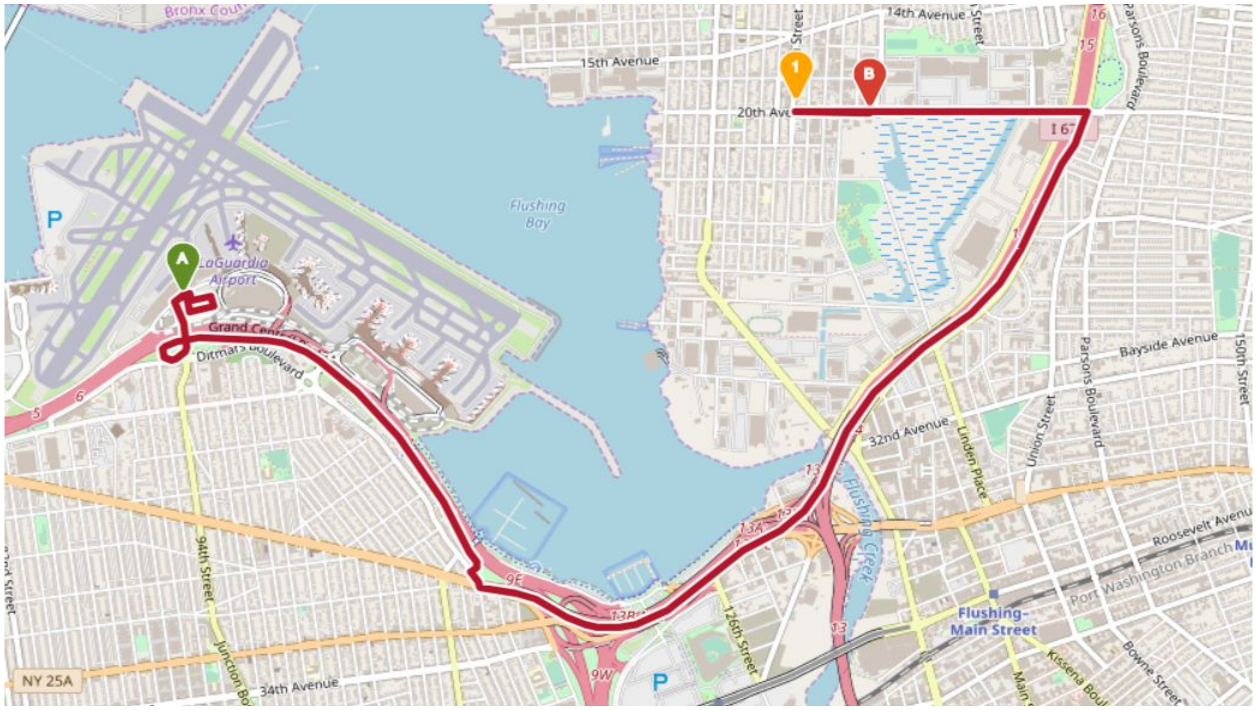
1.



In the above figure, the green point A is the source at LaGuardia and the other 2 points (orange point 1 and red point B) are the 2 destination points. The route highlighted indicates the actual shortest route that would be taken to travel to these

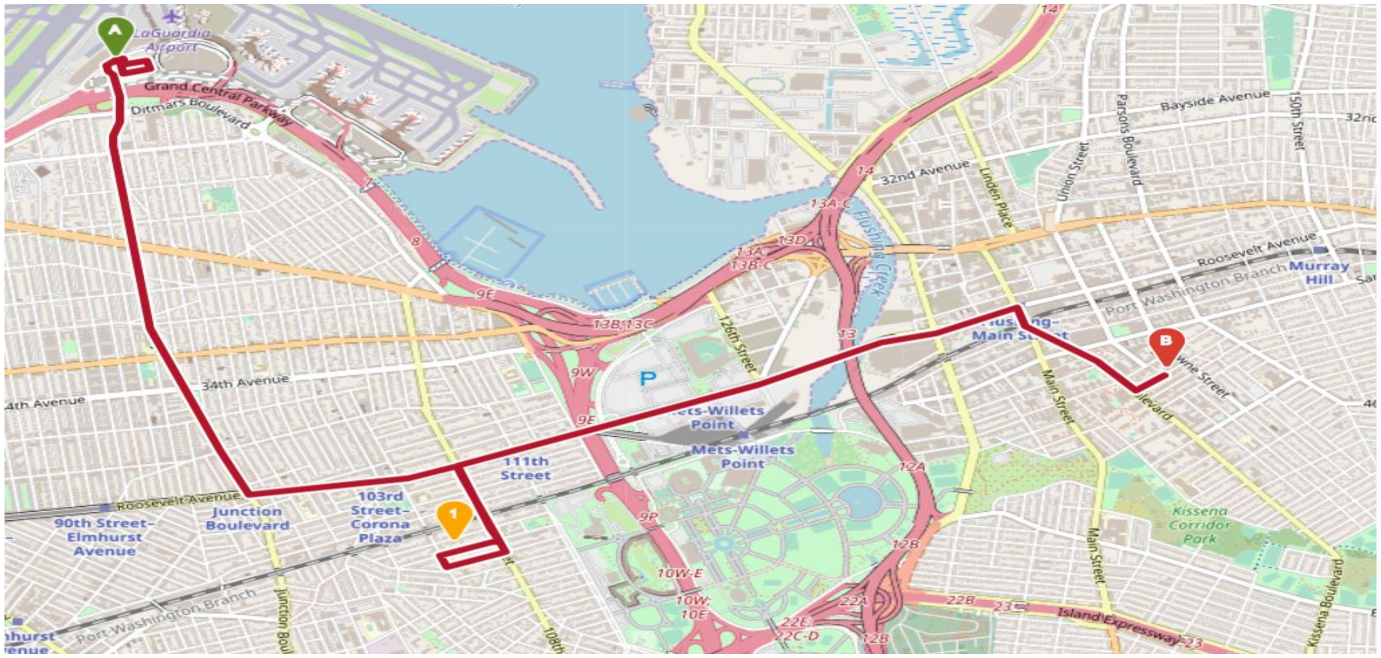
2 destination points from LaGuardia. As expected, we found that the rides were shareable and that passenger 1 with the orange destination point 1 will be dropped first and the ride would continue on to drop passenger 2 at B, as the .

2.



Similarly, for this example, passenger 2 with the red destination point B was dropped first and then passenger 1 at the orange point 1.

3.



Lastly for this example, the trips were not shareable as destination point 1 was way out of the route to destination point B, possibly violating the delay constraint of passenger 2.

6. Results

Results are obtained for 2 runs of 17 May, 2016(2nd run after fixing a bug), 1 run for the whole month of January 2016, 1 run for the 1st week of February 2016 and 1 rerun for January 2016 using the optimized approach discussed in the issues section.

The output for each case has the following plots:

1. Plot of Average Distance Saved as % of the Total Unshared Distance in each pool.
2. Plot of Average Number of Trips Saved as % of the Total Number of individual Trips in each pool
3. Plot of the runtime in seconds for each pool.

For each plot, the X-axis is for the pool size of 5 minutes and pool size of 10 minutes.

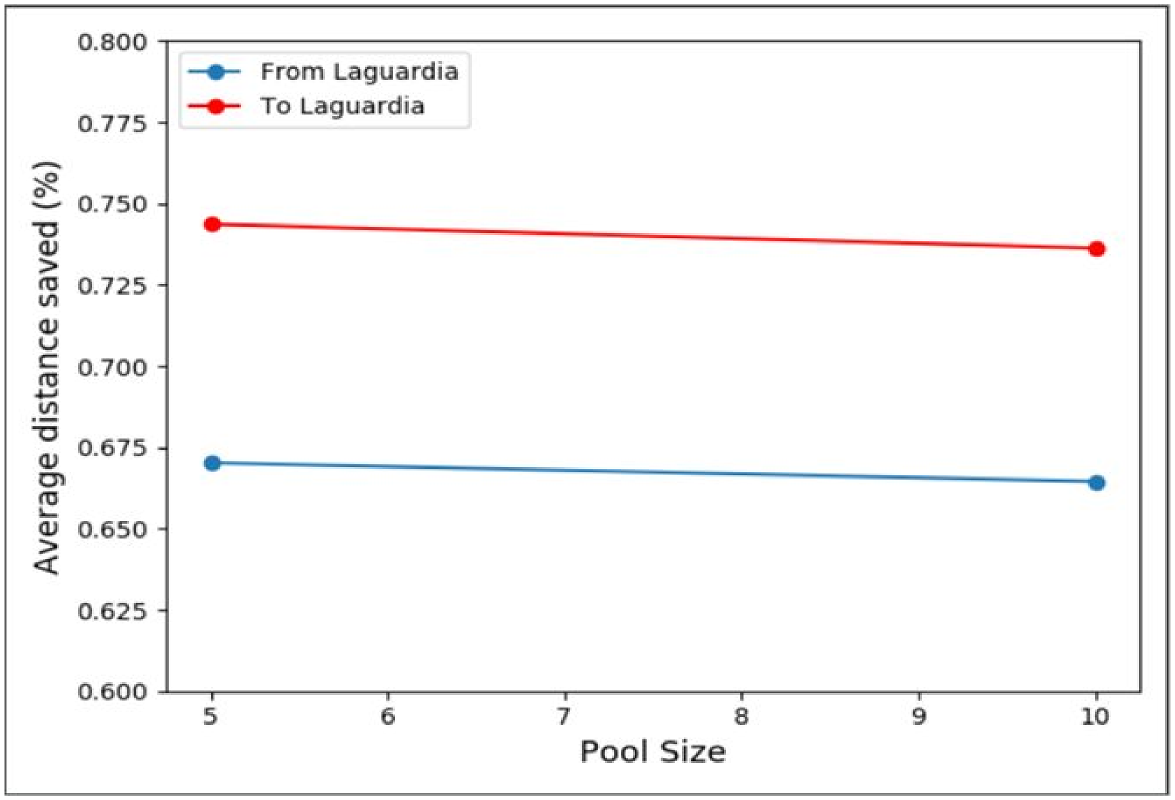
Note that for each plot having distance saved and trips saved, the y-axis values are represented as fractions:
So a 0.45 on the y-axis should be interpreted as 45%, 0.65 as 65%, and so on.

6.1 Results of initial run for 17 May, 2016:

12796 trips from LaGuardia

5792 trips to LaGuardia

Distance Saved (%)



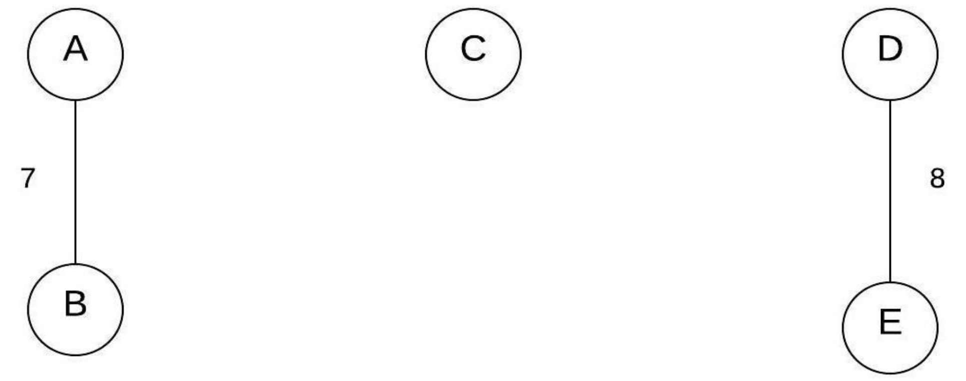
Explanation:

3 problems can be observed in the plot shown above:

1. Distance Saved is significantly higher than expected (in the range of 65%-75%).
2. Distance Saved reduces from the 5 minute pool window to the 10 minute pool window instead of increasing.
3. Distance Saved for trips to LaGuardia is higher than distance saved for trips from LaGuardia for both the 5 minute and 10 minute pools.

This problem arises because of the formula initially used to calculate the distance saved per pool, as explained below.

Consider the following ride sharing graph from above after the maximum matching algorithm has finished matching trips - A and C are shared, D and E are shared, and C is an individual trip:
Let the total unshared distance be 150 miles.

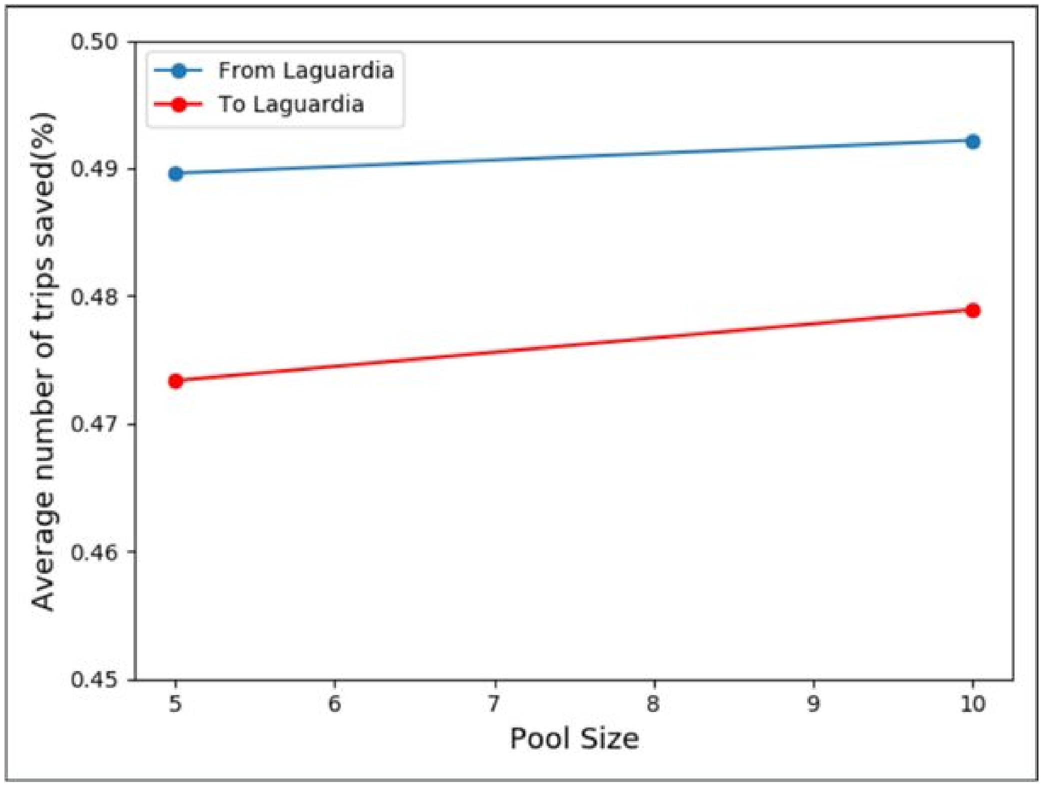


Total Distance saved is 15 miles.
Correct Computation for the distance saved (%) per pool would be:
Distance Saved = (Total distance saved) / (Total Unshared Distance)

= 15 / 150
= 10%

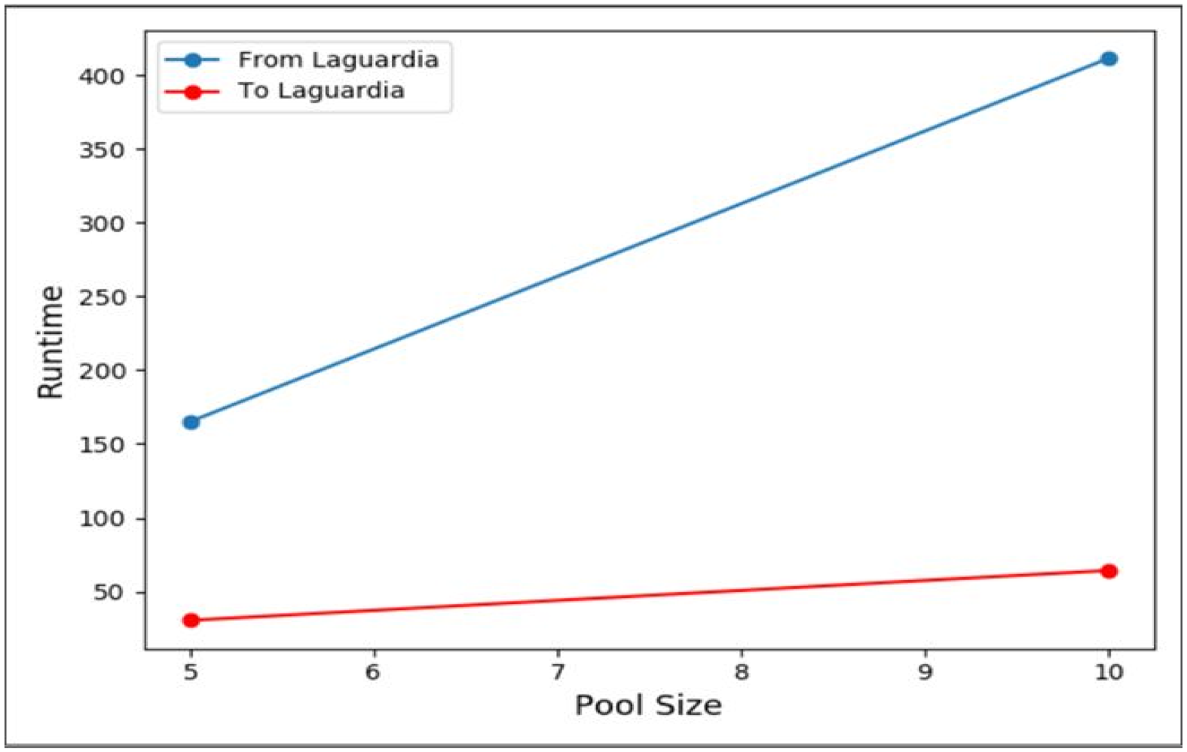
Initial Incorrect Computation for the distance saved (%) per pool used:
Distance Saved = (Total Unshared Distance - Total distance saved) / (Total Unshared Distance)
= (150 - 15) / 150
= 135 / 150
= 90%
Because of the 2nd formula used, the distance saved was higher up in the range of 65% - 75% and decreased from the 5min pool window to the 10min pool window.

Trips Saved (%)



Explanation:
The 10 minute pool window will have more number of trips to share, so as expected, the average % of trips saved per pool increased from the 5 minute pool window to the 10 minute window.

Runtime (seconds)



Explanation :
The runtime depends on the number of trips per pool, as every pair of trips in a pool is considered for sharing. Similar to the other plots, the runtime increases from the 5 minute window to the 10 minute window.

The average number of trips per pool for both pool sizes are as follows:

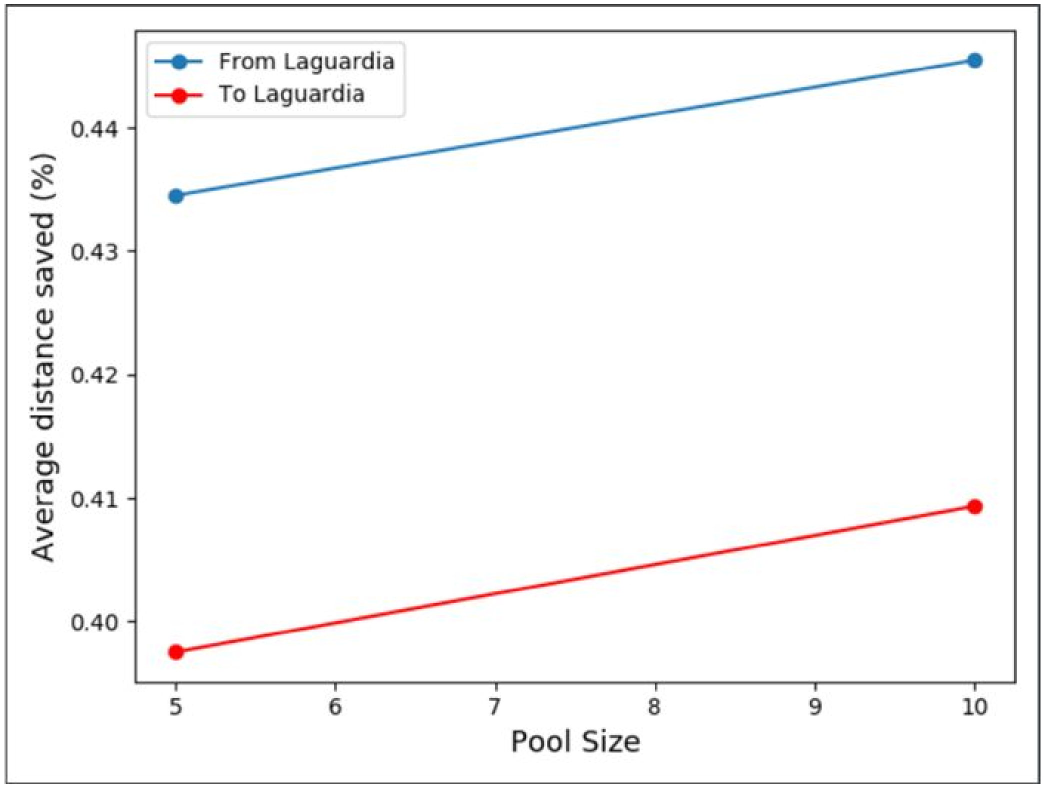
- Trips from LaGuardia:
  - 5 minutes Pool: 61 trips per pool on average
  - 10 minutes Pool: 107 trips per pool on average
- Trips to LaGuardia:
  - 5 minutes Pool: 27 trips per pool on average
  - 10 minutes Pool: 46 trips per pool on average

Thus the average number of trips per pool for the trips from LaGuardia is significantly higher than the same for the trips to LaGuardia.

This explains the large gap between the blue and the red lines in the runtime plot.
Also the difference in the average number of trips per pool across different pool sizes is much larger for the trips from laguardia, thereby explaining the larger slope of the blue line.

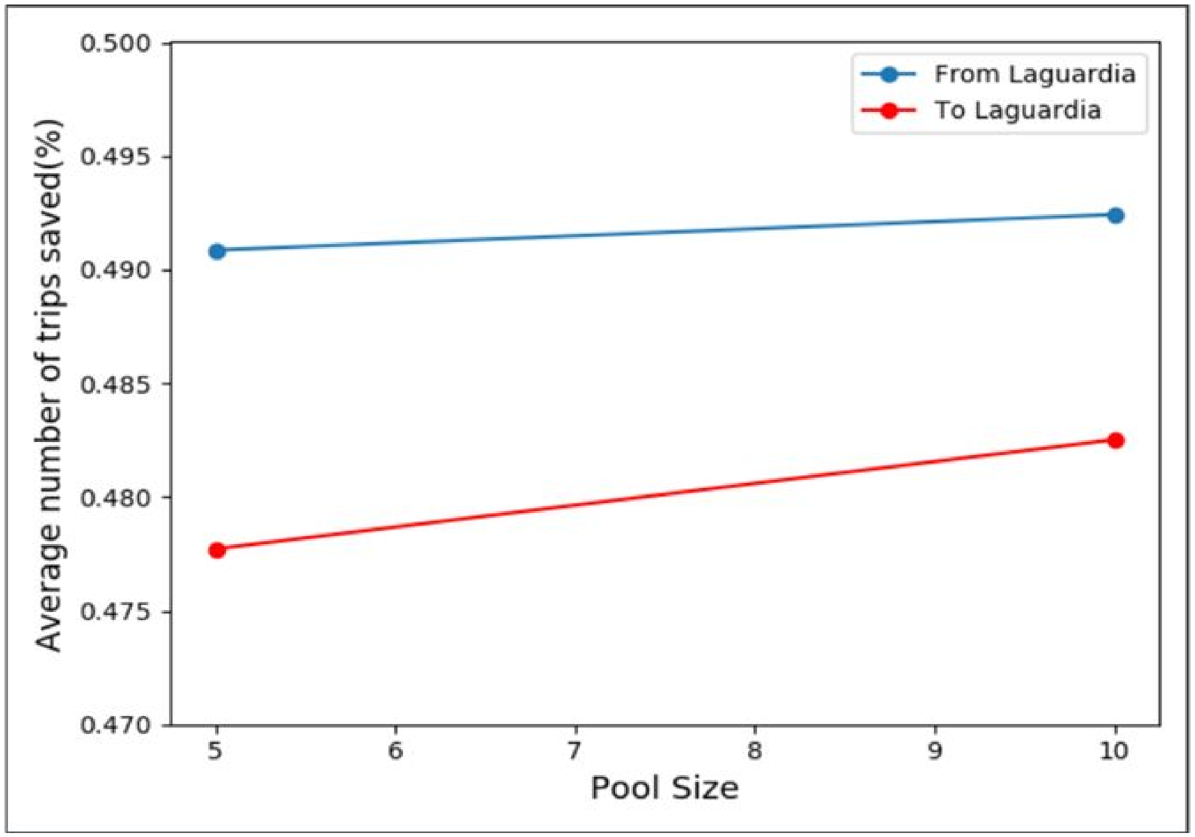
6.2 Results of rerun for 17 May 2016 (After correcting the formula used to compute the % of distance saved per pool)

Distance Saved (%)



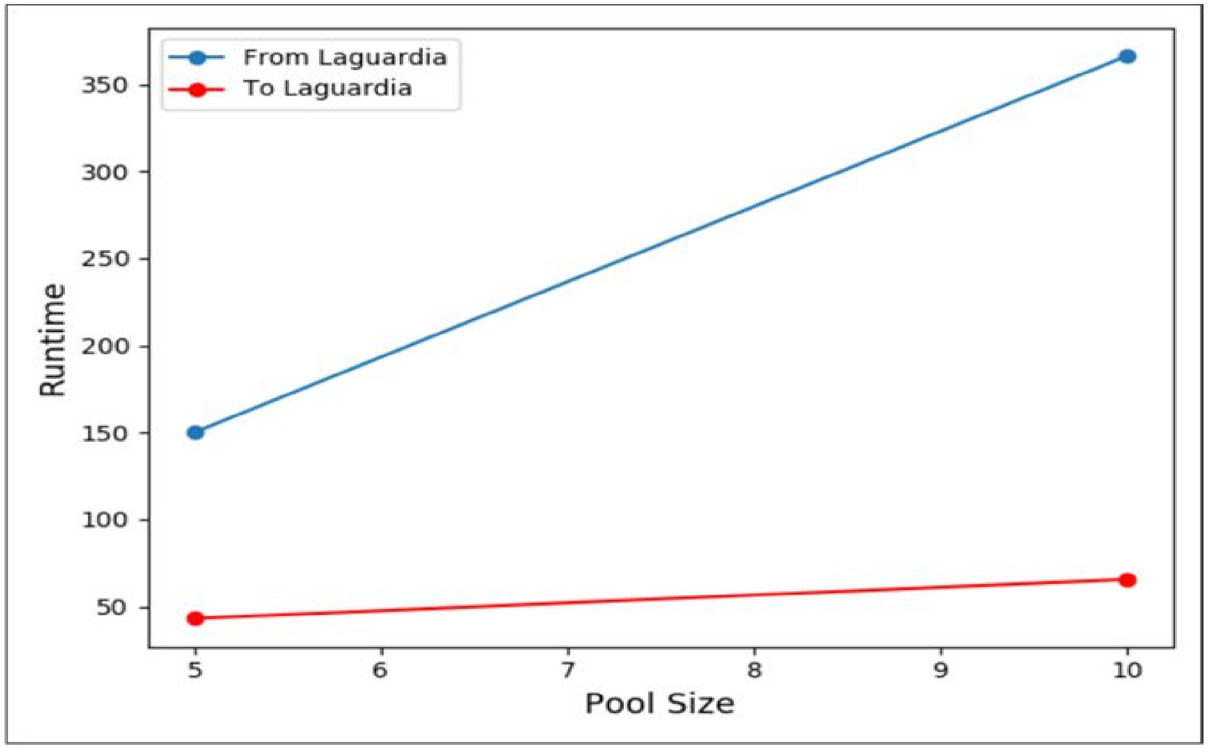
Here, as expected the distance saved is in the range of ~35%-45%, which is consistent with that of the other test dates considered, there is an increase from the 5 minute pool to the 10 minute pool window and the distance saved for trips from LaGuardia is higher than that for trips to LaGuardia for both the pool window sizes.

Trips Saved (%)



Trips saved is almost the same as for the initial run.

Runtime (seconds)

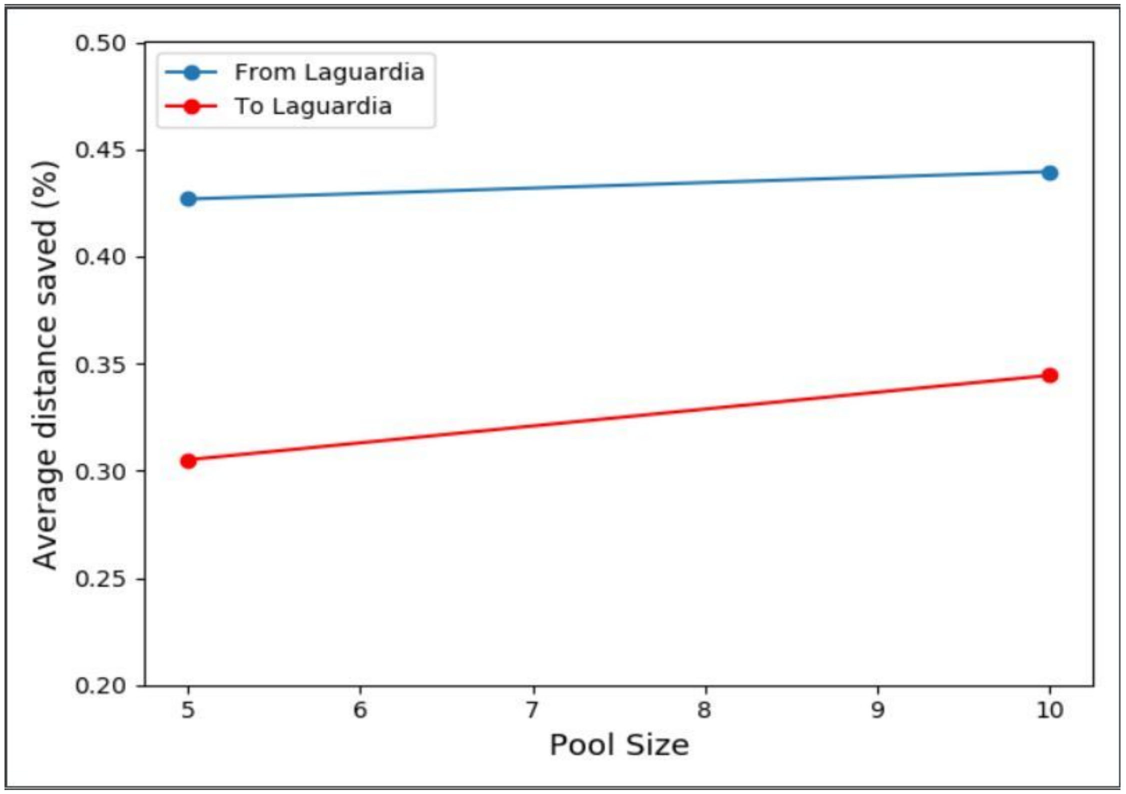


Runtime is the same as for the initial run.

6.3 Results for January 2016 (Whole month)

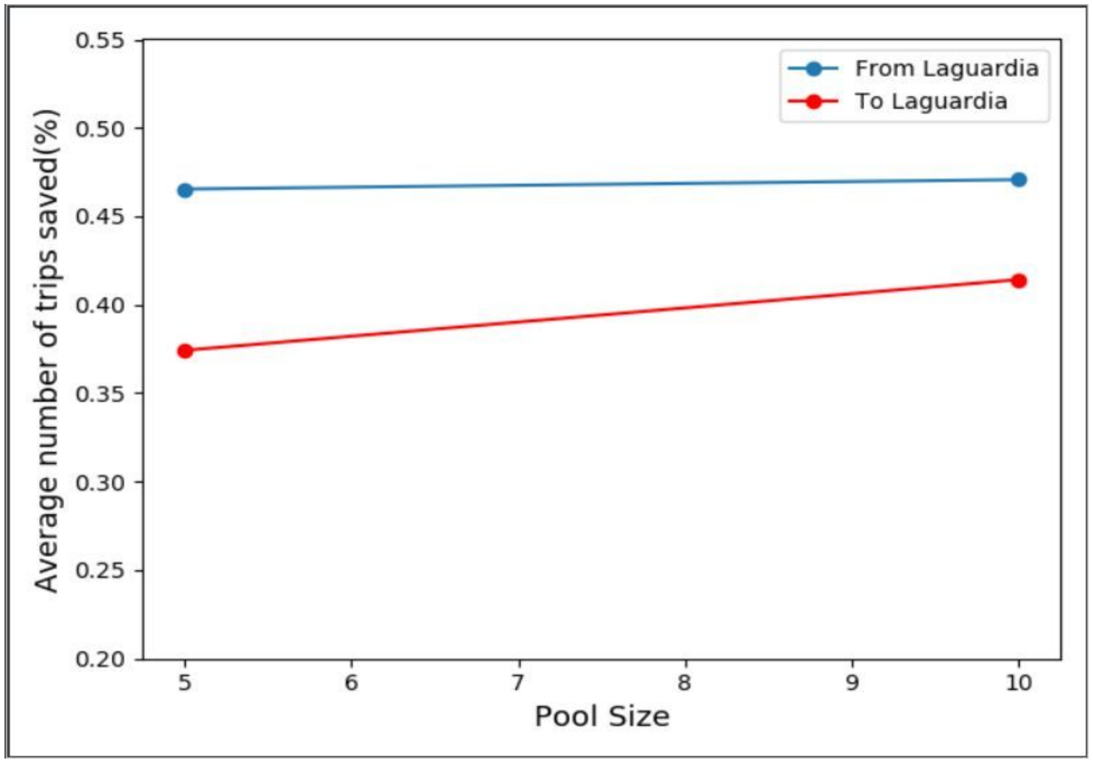
28100 trips from LaGuardia
11060 trips to LaGuardia

Distance Saved (%)



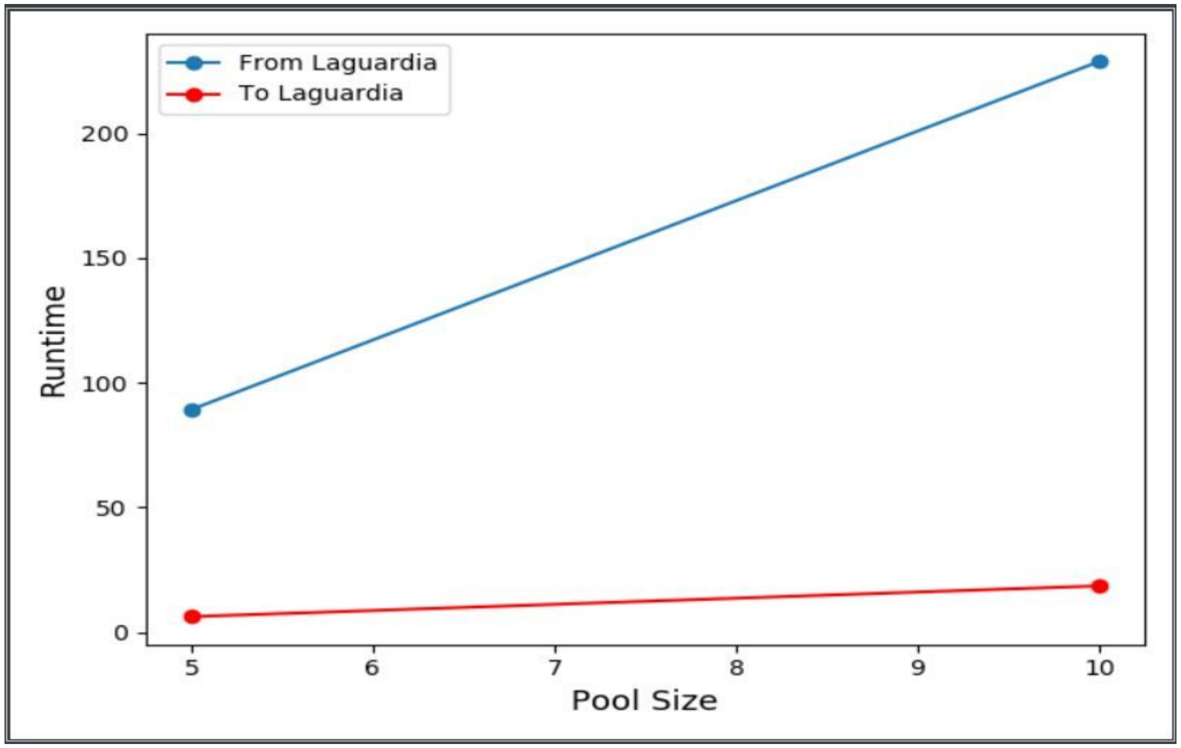
Similar to the results of the rerun for 17 May 2016, the distance saved is in the range of ~30%-45% and increasing from the 5 minute pools to the 10 minute pools.

Trips Saved (%)



For the trips from laguardia, the proportion of trips saved with respect to the total number of trips in a pool is almost the same for both the 5 minute(46.5%) and 10 minute(47.1%) pools, so the blue line in the plot has a very small slope.

Runtime (Seconds)



The runtime for processing the trips from LaGuardia is much higher than that for the trips to LaGuardia, as we have the following:

Trips from LaGuardia:

- 5 minutes pool : 37 trips per pool on average
- 10 minutes pool : 65 trips per pool on average

Trips to LaGuardia:

5 minutes pool: 14 trips per pool on average
10 minutes pool: 24 trips per pool on average

As the number of trips per pool for trips to LaGuardia is significantly smaller than the number of trips per pool for trips from LaGuardia, we observe the same effect on the runtime.

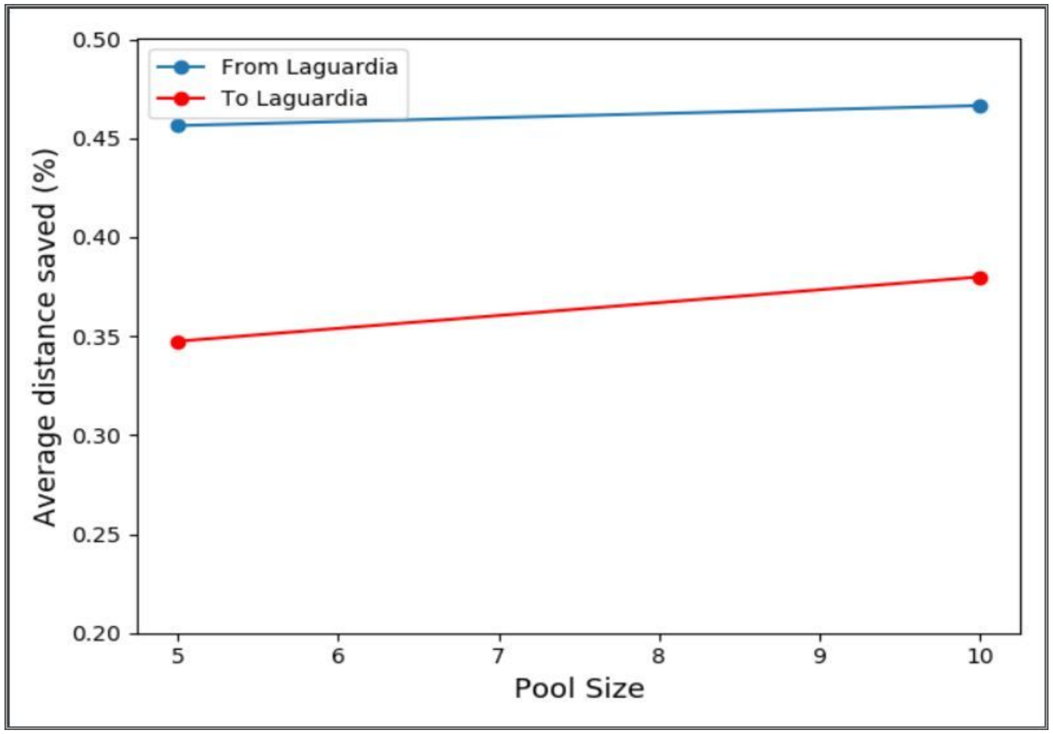
Also the difference in the number of trips across pool sizes is larger for the trips from LaGuardia(28 vs 10), so the blue line for the same has a much higher slope than that of the red line.

6.4 Results for 1st week of February 2016:

56205 Trips from LaGuardia

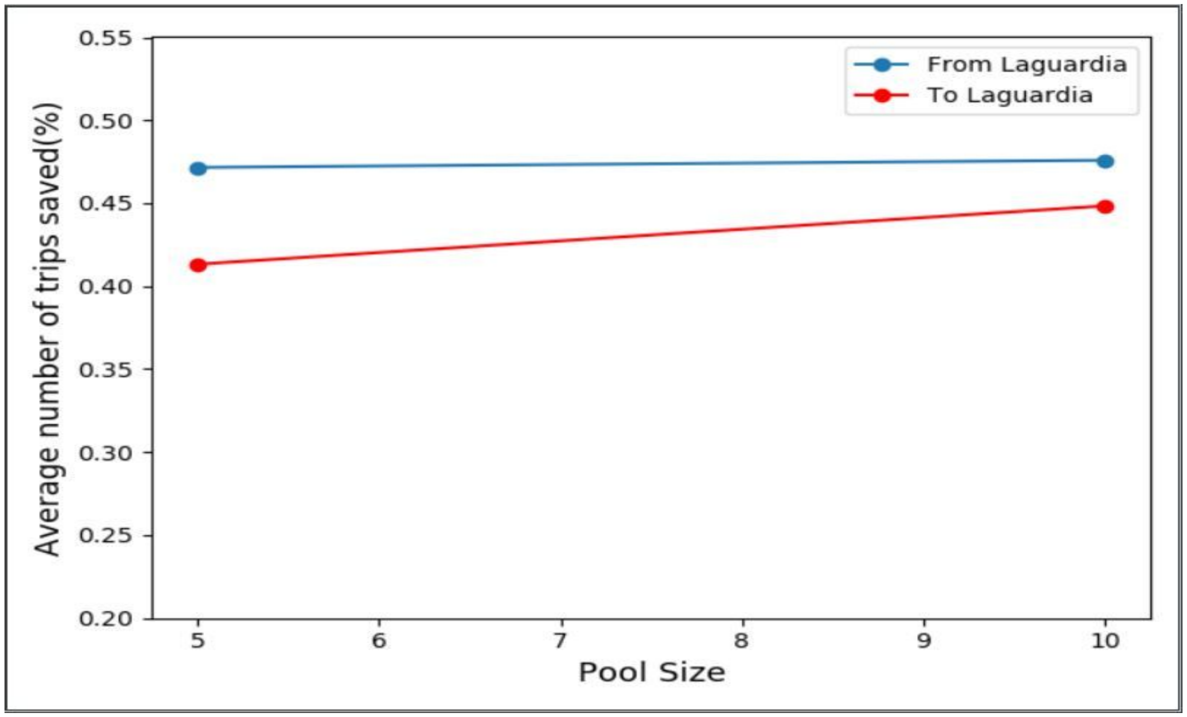
28918 Trips to LaGuardia

Distance Saved (%)



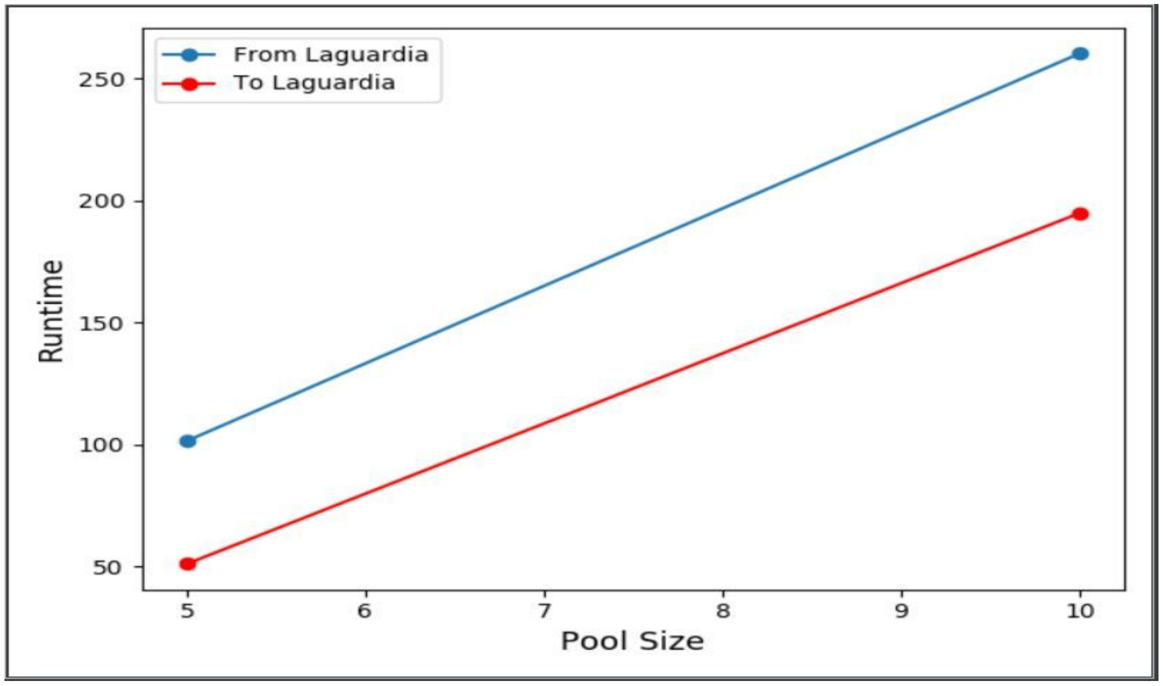
Distance Saved is as expected for both the pool windows.

Trips Saved (%)



Though there is an increase in the amount of trips saved from the 5 minute pool window to the 10 minute pool window for both ways (to/from LaGuardia), this increase is only marginal for the trips from LaGuardia (47.1% to 47.6%), producing the blue line with a very small slope.

Runtime (seconds)



Both the lines have a large slope in this case. Similar to the other plots, we present the number of trips per pool below:

Trips from LaGuardia:

5 minutes pools : 42 trips per pool on average

10 minutes pools: 73 trips per pool on average

Trips to LaGuardia:

5 minutes pools: 20 trips per pool on average

10 minutes pools: 36 trips per pool on average

Given the difference in number of trips per pool across the 2 pool windows for both cases, the slopes of both the lines are still larger than expected. This is because for some of the pools, the runtime reaches a value of ~1000 seconds, when the number of trips in the pool exceeds 150. These outlier values then increase the average runtime for the whole pool window.

7. Issues and Drawbacks

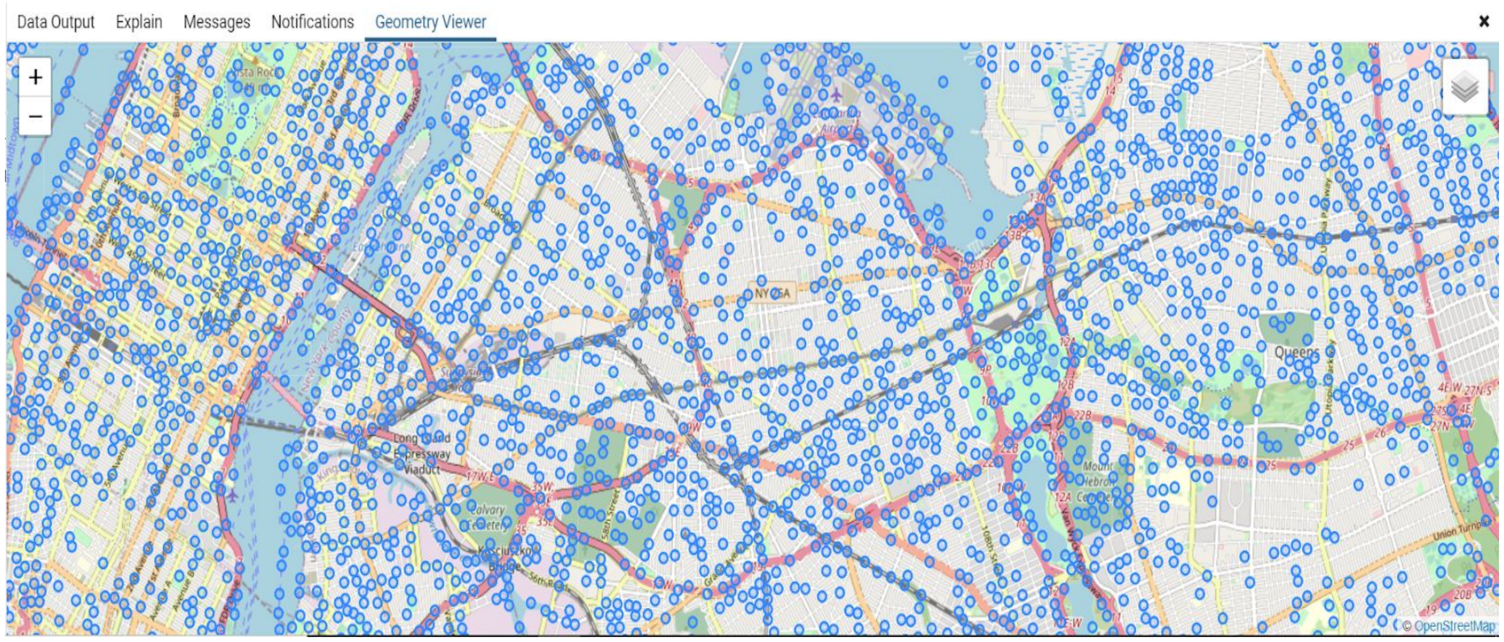
- **Average distance error accrued**:
  Because of the threshold value of 100 metres used to filter out and discretize the points available for precomputation and to reduce the search space and time, some source or destination points will be mapped / approximated to a very far away point if there are no points available in that area where the source or destination point is located, thus increasing the distance error for that point.
  So for each pool of both the January and February Trips, the distance between the actual and the approximated point for each trip considered is calculated and averaged. The values are presented below:

January 2016:

Pool	Average Distance Error
5 minute Pool(From LaGuardia)	~31km
10 minute Pool (From LaGuardia)	~24km
5 minute Pool (To LaGuardia)	~25km
10 minute Pool (To LaGuardia)	~40km

February 2016:

Pool	Average Distance Error
5 minute Pool(From LaGuardia)	~18km
10 minute Pool (From LaGuardia)	~21km
5 minute Pool (To LaGuardia)	~26km
10 minute Pool (To LaGuardia)	~29km



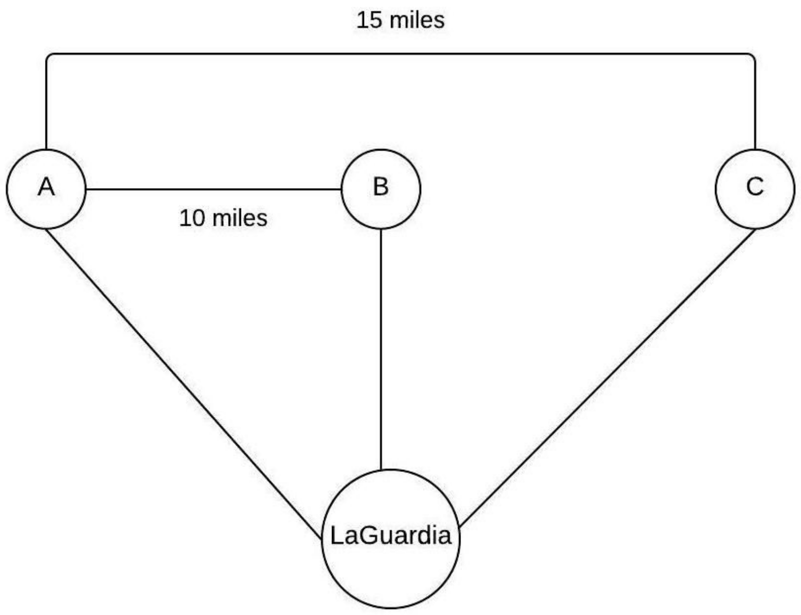
In the above part of the map of New York, though most of the actual source and destination points will be mapped to the nearest points (blue points) only several metres away, there are clearly some gaps/empty areas in the above map in the bottom and the top right areas that do not have any points.

If a trip source or destination point falls in one of those empty areas then possibly it's nearest mapped point will be several kilometres away, thus increasing the average distance error by a large value as shown in the tables above.

A straightforward solution to this problem would be to reduce the threshold used i.e. 100 metres to a small value such as 10 metres, recompute and redistribute the points across the whole of New York (Manhattan, Brooklyn, Queens, etc.) making the map denser. This will decrease the distance error but will also increase the search time to find the nearest point for each trip and increase the size of the points available for precomputation.

- **Large Runtime**
  The runtime in most of the previous plots peaks upto ~250-275 seconds on average. This is because in the initial algorithm implementation, every possible combination of trips inside a pool is processed explicitly to evaluate the shareability conditions for the no-walking and walking-case.
  We try to bring down the maximum value that the runtime can reach.
  Here the walking case also involves going through each of the drop points which are associated with the destination under consideration, which adds an extra overhead.
  This latter case of iterating through every drop point in the walking case is avoided in some cases using the following idea explained below:

  Consider 3 trips A, B, C with corresponding destinations A, B, C.



Distance between A and B is 10 miles and distance between A and C is 15 miles.
In the current pool, let’s say we are iterating over every trip other than A to check whether those trips are shareable with A or not, which was a straightforward brute force idea used initially.

Let’s assume that A and B are not shareable. Now we move onto checking trip A with trip C.

C is farther away from A than B is (from the above figure), so it is highly unlikely that A and C will be shareable given that A was not shareable with B, even if C has a higher delay than that of B. So we do not process trip C at all, and move onto the next trip i.e. D.
This clearly avoids fetching the dropoff locations for C from the database and iterating over and evaluating the shareability conditions on every drop point of A and that of C, which can be as high as 100 drop points walkable within 10 minutes from a given point, as given to us by the OpenRouteService API isochrones service.

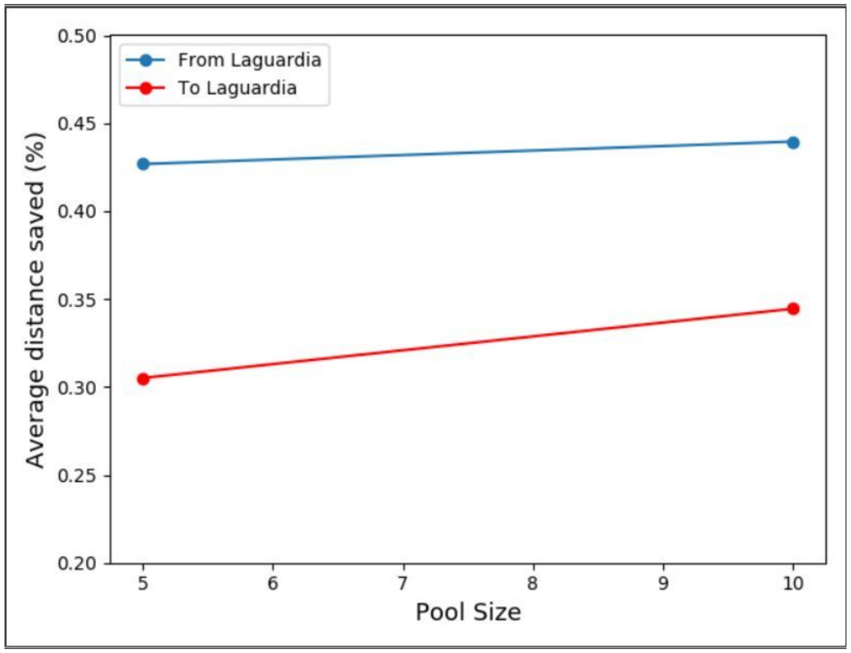
This requires keeping track of the minimum distance between a trip A and another trip which was found to not be shareable with A. In the above example, this minimum distance value will be 10 miles.

While checking some other trip C with A, we compare their distance with the minimum_non_shared_distance of A and if C’s distance from A is more than that minimum distance then we simply discard C and move onto the next trip.
In the above example, the distance between C and A is 15 miles, which is greater than the distance of 10 miles between trip A and B(which were not shared), so we discard C.

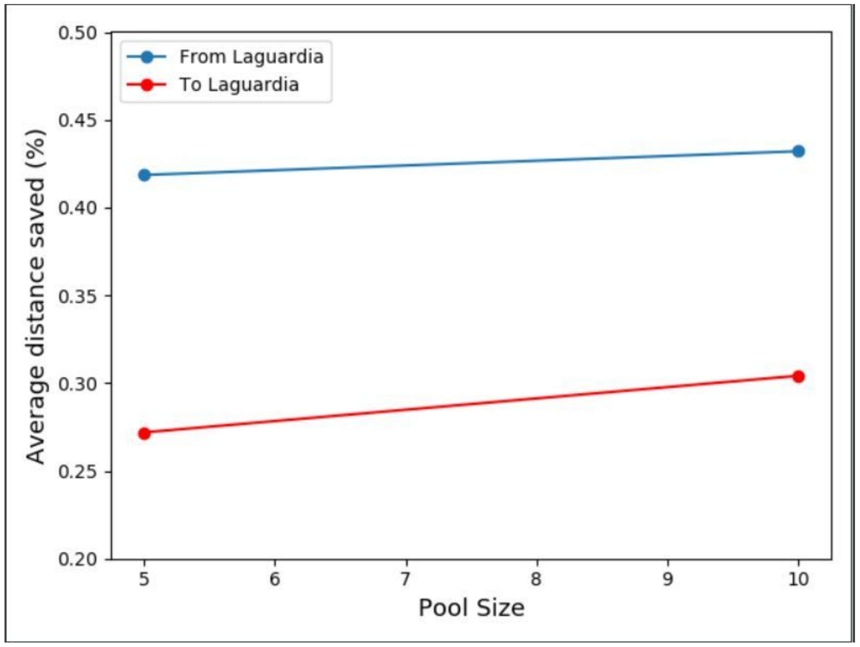
This solution was implemented as a possible optimization for reducing the runtime and the procedure was rerun for all trips for the whole month of January 2016. We compare the previous original January 2016 results with the runtime optimized results in the next page:

Comparison of original and optimized results for January 2016

January 2016 % Distance Saved (Original)

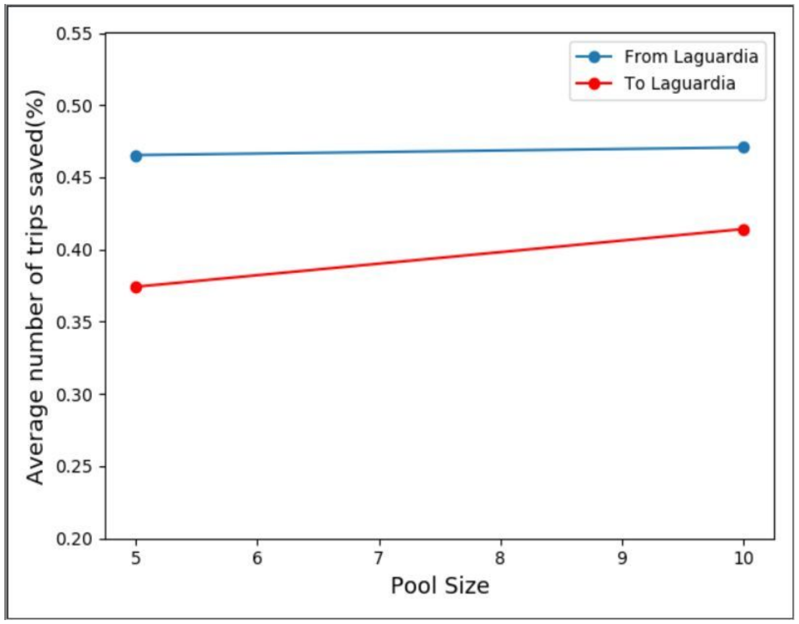


January 2016 % Distance Saved (Optimized)

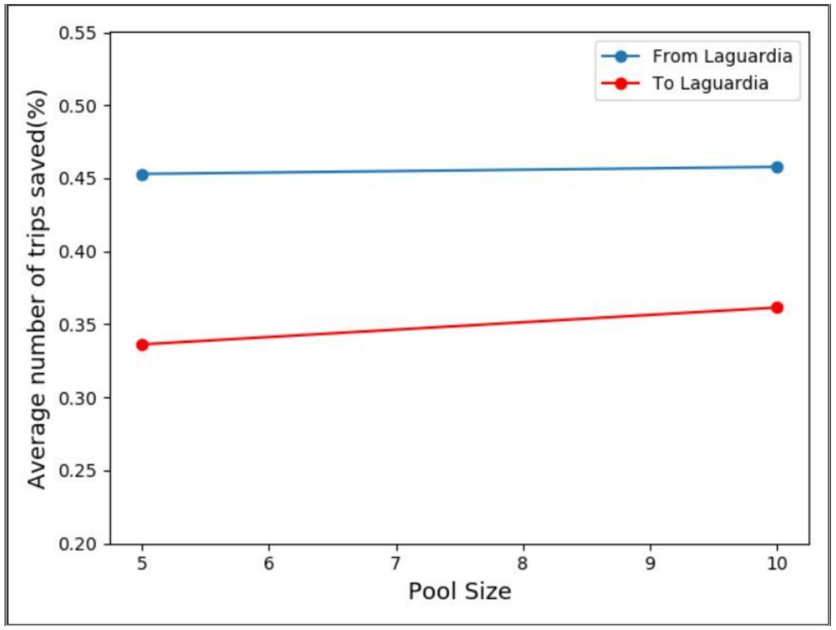


Explanation:
As observed, the distance saved does not vary for the trips from laguardia. There is only a slight decrease in the % distance saved for trips to laguardia in the optimized version, but the increase from the 5 minute pool window to the 10 minute pool window is almost the same.

January 2016 % Trips Saved (Original)



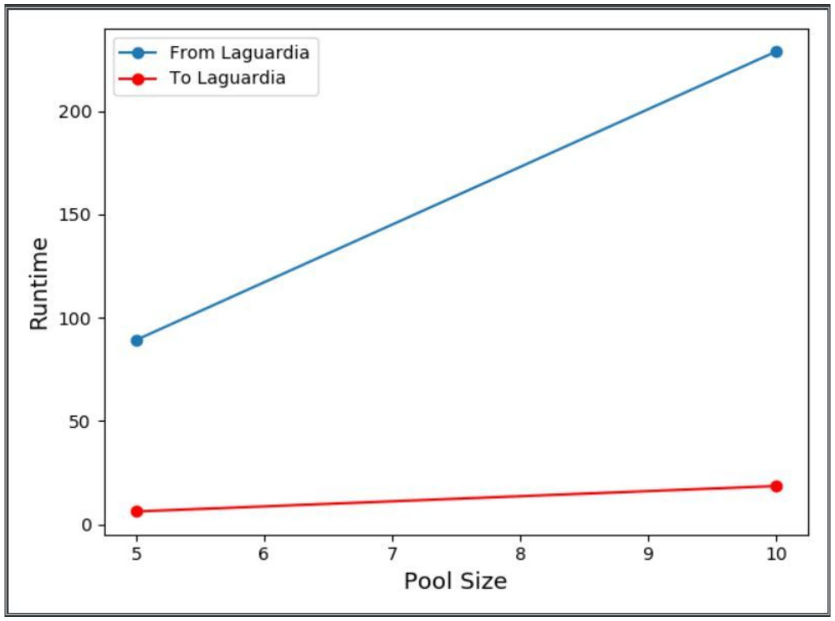
January 2016 % Trips Saved (Optimized)



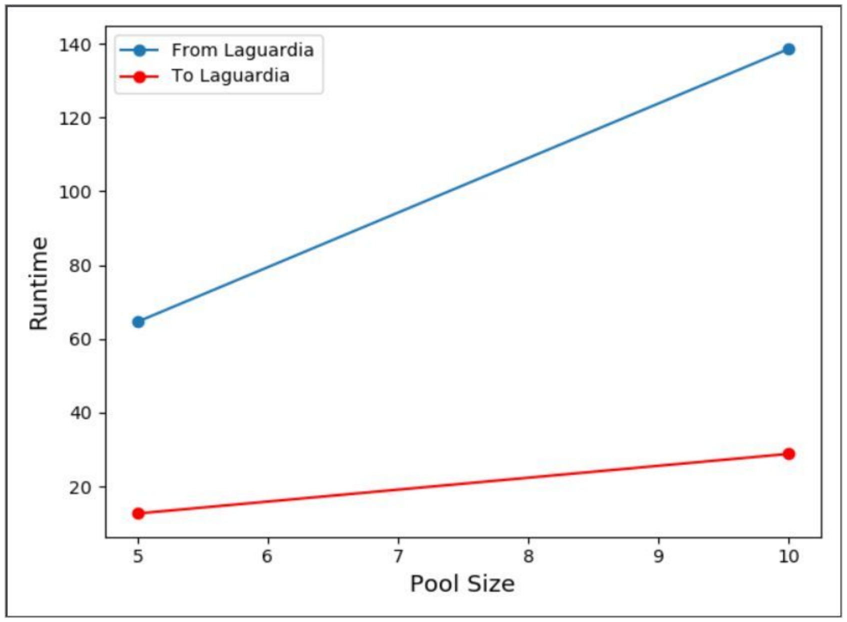
Explanation:

Similar to the distance saved, there is minimal variation in the results of the optimized version. Also, like the distance plots, there is a slight decrease in the % of trips saved in the optimized version for trips to laguardia but this decrease is minimal.

January 2016 Runtime in seconds(Original)



January 2016 Runtime in seconds(Optimized)



Explanation:

This is the most important plot comparison of the 3 pairs of plots presented. There is a significant improvement in the runtime of the trips from laguardia (~90s to 250s vs ~60s to 140s) with minimal variation in the runtime of trips to laguardia, indicating that the optimized approach for processing trips works well when we have a large number of trips to process in a given pool and we are considering walking(only for trips from laguardia), with minimal loss of accuracy compared to the straightforward brute force approach.

8. Conclusion

The following conclusions can be drawn based on the results generated by our experiments and execution for the different data sets.

- The number of trips in the ride sharing model reduces drastically as compared to all the trips taken individually. Based on most of the experiment results, we get a savings of ~40%-45% in the number of trips taken when trips are shared.

- Due to reduction in the number of trips, there were significant savings in terms of the distance travelled as well. Approximately 35%-40% of the total distance (total distance of individual trips) was saved when trips are shared.

- Most of the running time of the algorithm was consumed by processing every combination of trips in a pool and considering every drop point for both destinations under consideration. This runtime was reduced substantially for the trips of January 2016 from LaGuardia using the optimized approach explained above.

- The average distance error accrued was because of the empty areas produced with the final set of discretized points computed using a threshold of 100 metres between every point. This error can be reduced by reducing the threshold distance from 100 metres to a small value such as 10 metres, at the cost of increased point search time and storage required.

9. Collaboration & Project Management

Phase of Project	Task done by	Completed by
Algorithm Design	Apurva, Nandana, Siddhanth and Ashwin	February 9, 2020
Data Filtering	Nandana and Ashwin	February 28, 2020
Implementation	Apurva and Siddhanth	March 13, 2020
Testing	Apurva, Nandana, Siddhanth and Ashwin	April 5, 2020
Output plots	Siddhanth	April 26, 2020

10. References

- Open Street Map [openstreetmap.org]
- PostGIS [PostGIS — Spatial and Geographic Objects for PostgreSQL]
- OpenrouteService [openrouteservice.org]
- Efficient Algorithms for Finding Maximum Matching in Graphs" by Zvi Galil, ACM Computing Surveys, 1986