Resource Search

Project Report

Team:

Bhargav Arisetty
Harsha Muttavarapu
Reena Mary Puthota
Saket Sahasrabuddhe

TABLE OF CONTENTS

(use links to navigate through topics)

I. PROBLEM DEFINITION

The problem is defined in the context of crowdsourced taxicabs searching for customers to pick up. The system consists of four types of entities: a road network, mobile agents (i.e., taxicabs), and stationary resources (i.e., customers), and an assignment authority.

Agents are introduced to the system at once at the beginning of the operation, each located at a random location on the road network. The set of agents is fixed throughout the operation, the size of which will be referred to as the agent cardinality.

Resources are introduced to the system in a streaming fashion1 , each with a destination. Each resource has a maximum life time (MLT) starting from its introduction, beyond which the resource will be automatically removed from the system, an outcome which we will call resource expiration. It is possible that multiple resources are introduced at a single time unit. After an agent is introduced to the system, it is labeled as empty, and cruises along a path decided by the solution, which we will call a search path. When a resource is introduced to the system, the agent that meets the following conditions is assigned by the assignment authority to the resource:

1. The agent is empty;
2. The agent is closer (by shortest travel time) to the resource than any other agent;
3. The shortest travel time from the agent to the resource is smaller than the resource's remaining life time (i.e., MLT minus the time duration since the resource is introduced until the present time).

Once an agent is assigned to a resource, the agent is labeled as occupied and the resource is removed from the system. Then the agent moves to the resource (for pick-up) and then to the destination of the resource (for drop-off), along shortest-travel-time paths. Once the agent arrives at the destination, it is labeled as empty. If there is no agent that meets the above conditions, then the resource remains in the system until an agent meeting the above conditions appears or the resource expires. The agent knows neither when and where the future resources will be introduced, nor any information regarding other agents.

The assignment authority has an optional software module called the data model, which contest participants can choose to implement for the contest. The data model is accessible to the agents, and can be used to represent resource availability patterns and predict future availability. The contest organizers will define a training dataset for participants to build the data model, prior to the system operation. The training dataset contains records of the times and locations at which resources became available in a road network during a past time period. The system will operate on a resource dataset which we will call a test dataset. The test dataset is a randomly chosen subset of the training data. For example, the training dataset covers a few years whereas the test dataset covers a random day of these years.

The contest looks at the following three aspects of performance, in order of priority:

1. A search time of an agent is the amount of time from when the agent is labeled as empty until it picks up a resource. An agent may experience multiple search times, each corresponding to one assignment.
2. The wait time of a resource is the period of time from its introduction until its pick-up or expiration.
3. The expiration percentage is the percentage of expired resources. In summary, the problem definition is as follows:

Input:

1. A road network (map);
2. A training dataset (list of resource locations and timestamps);
3. A test dataset (list of resource locations and timestamps);
4. The number of agents (i.e., agent cardinality) and their initial locations. The agent cardinality may be 5000, 6000, 7000, 8000, 9000, or 10000.

Output: Search path for each agent.

Objective: Minimizing (in order of priority) the average search time, average wait time, and expiration percentage

II. DATA EXPLORATION

UNDERSTANDING the OpenStreetMap data model of Manhattan: In OpenStreetMap, each feature is described as one or more geometries (think for instance of a river's shape) with attached attribute data (think of the river's name). Geometries are described with three different *elements*: *nodes*, *ways* and *relations*. Attributes are described as *tags* that can be part of a node, a way or a relation. Nodes are the equivalent of a point, ways are like lines that connect points and relations are collections of points or ways that represent a larger whole. This is easiest to understand with a couple of examples.

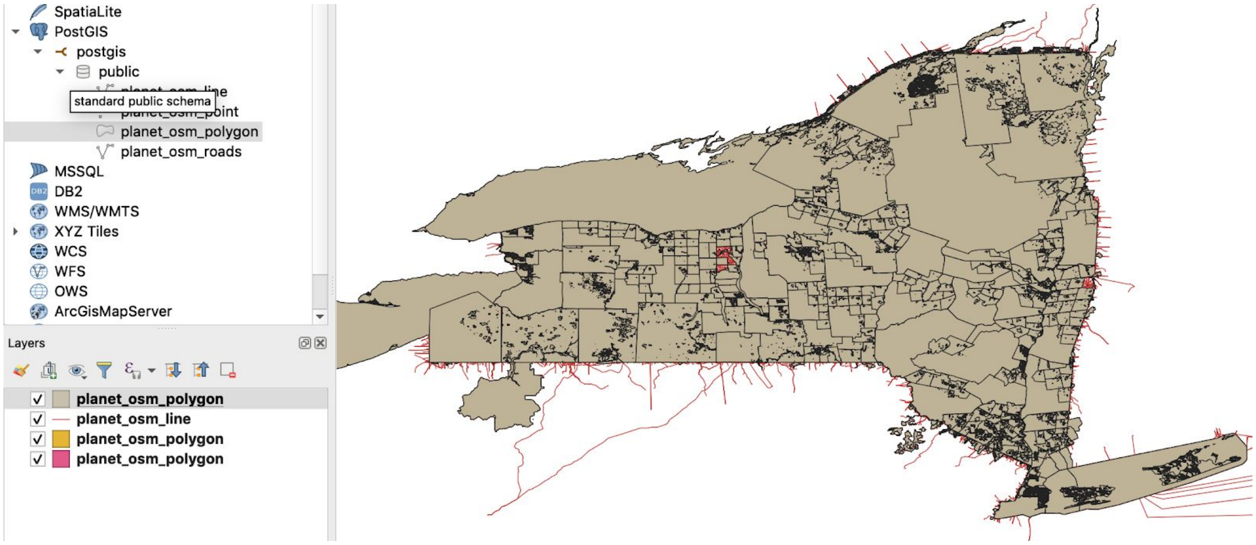
1. **Nodes:** Nodes are used to represent any kind of point type feature or just to designate a name for a point of interest in the vicinity. Here's a node representing a café:

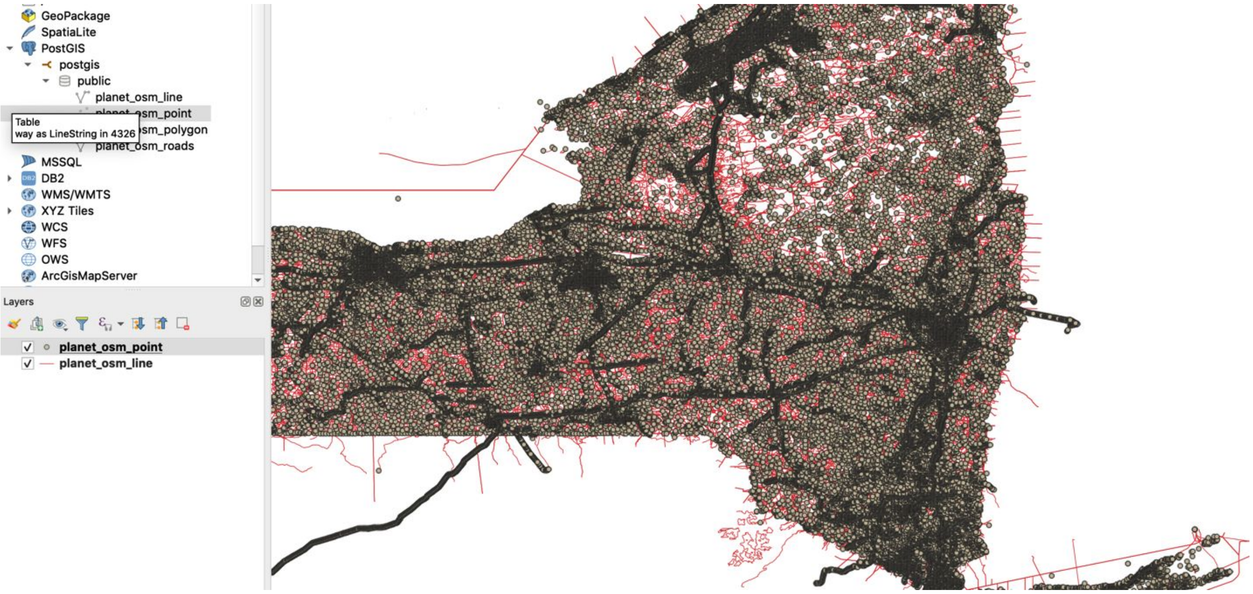
2. **Ways**: A way is a line feature *connecting two or more nodes* - like a road here:

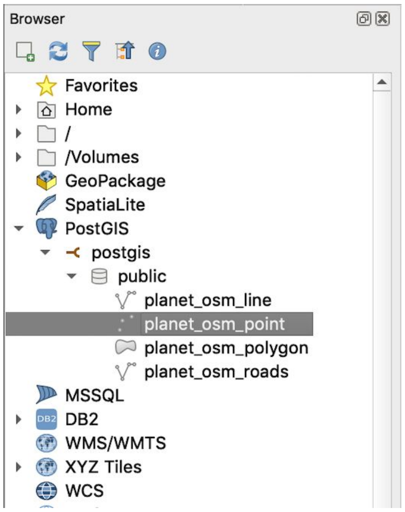
3. **Tags:** Any point type feature is a node. Whether the node designates a café, a school, a fire hydrant, a tree, a park, a mountain peak is entirely up to how the node is tagged. Any line type feature is a way. Whether the way is a road, a building, a lake, a railway, a cycleway is again, defined by how it is tagged. Tags can be on any element: on nodes, ways and relations.

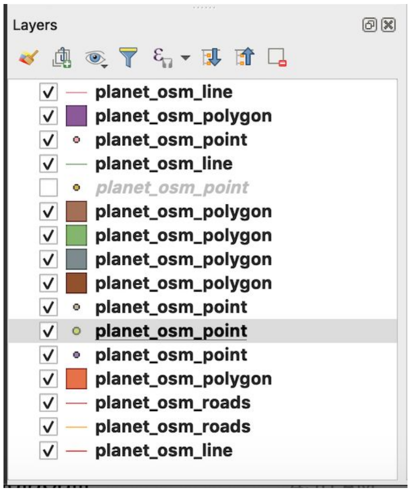
4. **Relations**: Relations are used to organize multiple nodes or ways into a larger whole.

DATA EXPLORATION USING QGIS: QGIS is a free and open-source cross-platform desktop geographic information system application that supports viewing, editing, and analysis of geospatial data.


SpatiaLite
PostGIS
postgis
public
planet_osm_line
standard public schema
planet_osm_point
planet_osm_polygon
planet_osm_roads
MSSQL
DB2
WMS/WMTS
XYZ Tiles
WCS
WFS
OWS
ArcGisMapServer
Layers
planet_osm_polygon
planet_osm_line
planet_osm_polygon
planet_osm_polygon


GeoPackage
SpatiaLite
PostGIS
postgis
public
planet_osm_line
planet_osm_point
Table
way as LineString in 4326
planet_osm_polygon
planet_osm_roads
MSSQL
DB2
WMS/WMTS
XYZ Tiles
WCS
WFS
OWS
ArcGisMapServer
Layers
planet_osm_point
planet_osm_line


Browser
Favorites
Home
/
/Volumes
GeoPackage
SpatiaLite
PostGIS
postgis
public
planet_osm_line
planet_osm_point
planet_osm_polygon
planet_osm_roads
MSSQL
DB2
WMS/WMTS
XYZ Tiles
WCS


Layers
☑ planet_osm_line
☑ planet_osm_polygon
☑ planet_osm_point
☑ planet_osm_line
□ planet_osm_point
☑ planet_osm_polygon
☑ planet_osm_polygon
☑ planet_osm_polygon
☑ planet_osm_polygon
☑ planet_osm_point
☑ planet_osm_point
☑ planet_osm_point
☑ planet_osm_polygon
☑ planet_osm_roads
☑ planet_osm_roads
☑ planet_osm_line

III. ALGORITHMS EVALUATED

III.A. INITIAL ALGORITHM

Cab/Taxi system forms important part of public transport system. Uber/Lyft drivers have an advantage of having a central authority. We needed to architect a solution with the following constraints

1. CONSTRAINTS

- There are no direct or indirect communications between agents.
- Search path of an agent is constrained to the road network and adheres to traffic limitations provided by the map.
- Agent knows neither when nor where the future resources will be introduced.
- No ride sharing allowed. Agent has to empty for assignment.
- Closest (travel time) agent is allocated (always).
- Closest agent's travel time should be less than resource's remaining life time.

2. GOALS:

To develop an algorithm which gives search path, abiding by above constraints and with the following optimizations

- To reduce idle search time for an agent (Tax/Cabs).
- To reduce wait time for a resource (Customer)
- To reduce number of expired requests (Customer)

3. INITIAL ASSUMPTIONS

- Every road/path has same speed limit viz 30 miles per hours.
- Every path/road is two way (U turns are allowed).
- No influence of traffic conditions, traffic stops etc.
- Herding effect will be reduced/eliminated by random probabilistic selection.

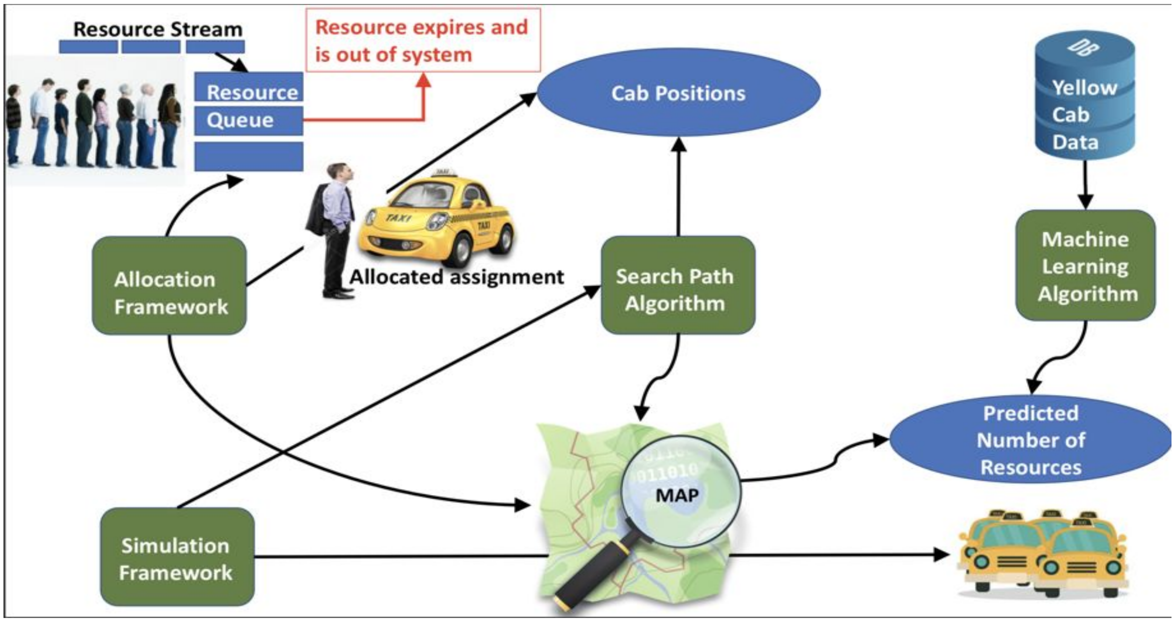
4. ALGORITHMS CONSIDERED:

Machine Learning:

1. Linear Regression
2. SVR(Support Vector Regression)
3. RANSAC(RANdom SAmple Consensus)

Search Algorithm:

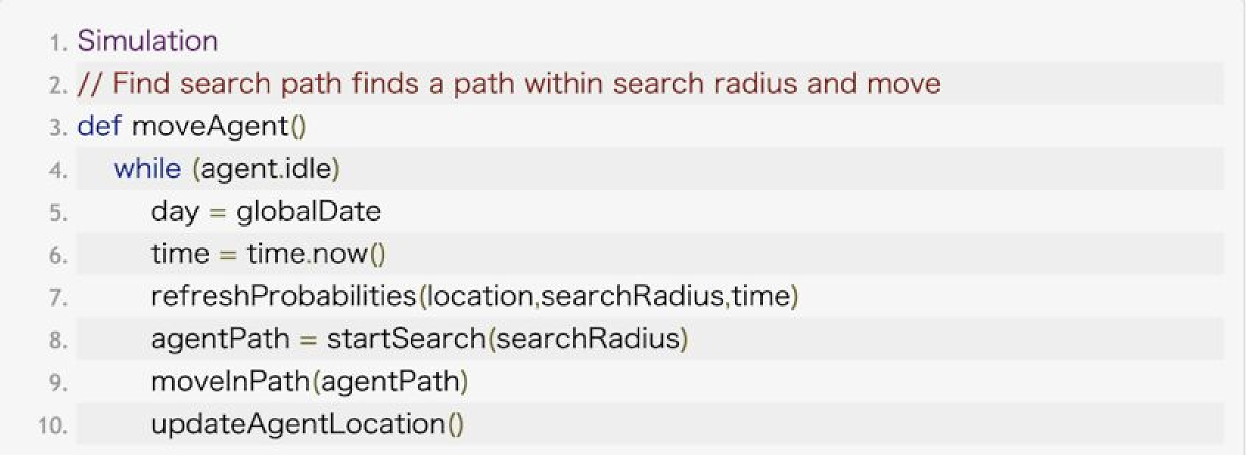
Reducing probabilistic model to shortest search path algorithm



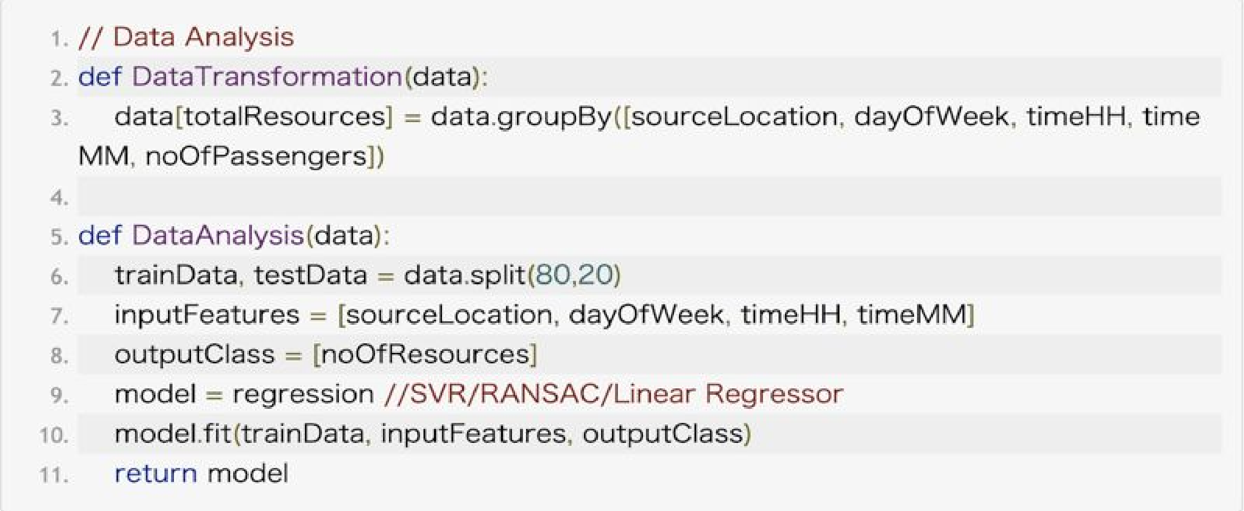
- Initially, we considered Probabilistic approach for path search.
- Let weight of a feasible path be the number of possible/expected resources for that particular path.
- Probabilities will be calculated for each path based on the number of resources of within the 1-mile (this is configurable) radius of the agent.
- The NYC 2016 cab ride data will be transformed and used for machine learning. We explored Linear Regression and SVR for the problem.
- Output of machine learning algorithm will be the NYC map graph with each edge having resource predictions based on features (time of the day, day of the week etc.).

5. ALGORITHM

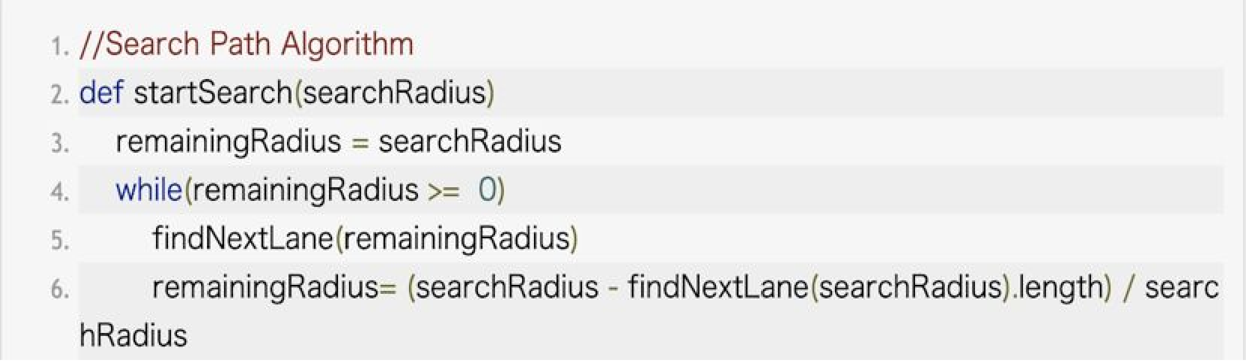
1. Group all the data by location, day of the week, time and no of passengers
2. Train the data on these features and return a model which is able to predict the number of resources at a location for the test data/new data
3. Move the cabs based on the probabilities generated by the expected number of resources at a given location at a given time from all possible paths within a given radius by refreshing and recalculating the probabilities at every 5-minute interval.
4. Allocate the closest cab to a resource, based on its remaining lifetime and the global resource expiry clock and the time taken for the cab to reach the resource.



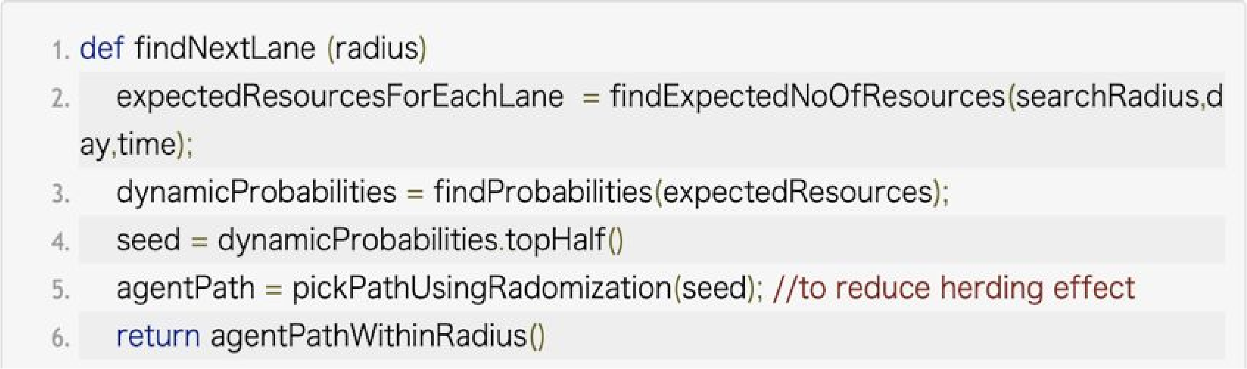
```
Simulation
// Find search path finds a path within search radius and move
def moveAgent()
    while (agent.idle)
        day = globalDate
        time = time.now()
        refreshProbabilities(location,searchRadius,time)
        agentPath = startSearch(searchRadius)
        moveInPath(agentPath)
        updateAgentLocation()
```



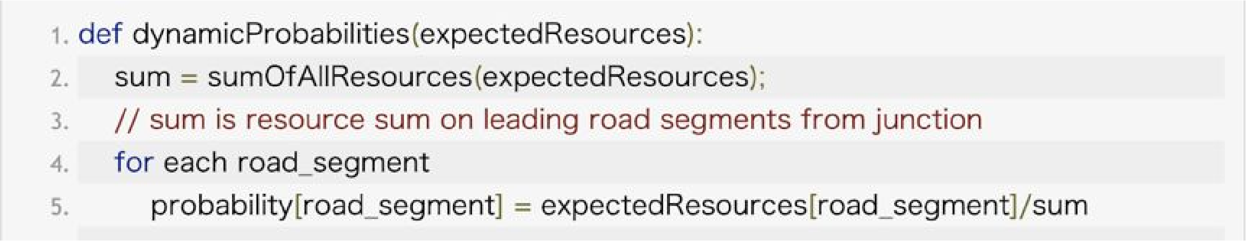
```
// Data Analysis
def DataTransformation(data):
    data[totalResources] = data.groupBy([sourceLocation, dayOfWeek, timeHH, timeMM, noOfPassengers])

def DataAnalysis(data):
    trainData, testData = data.split(80,20)
    inputFeatures = [sourceLocation, dayOfWeek, timeHH, timeMM]
    outputClass = [noOfResources]
    model = regression //SVR/RANSAC/Linear Regressor
    model.fit(trainData, inputFeatures, outputClass)
    return model
```



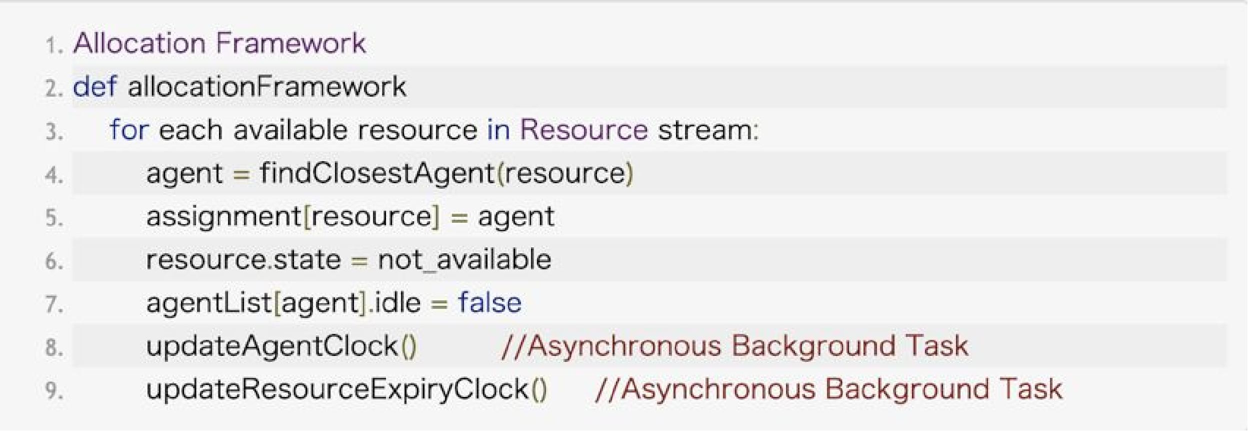
```
//Search Path Algorithm
def startSearch(searchRadius)
   remainingRadius = searchRadius
   while(remainingRadius >=  0)
      findNextLane(remainingRadius)
      remainingRadius= (searchRadius - findNextLane(searchRadius).length) / searchRadius
```



```
def findNextLane (radius)
   expectedResourcesForEachLane  = findExpectedNoOfResources(searchRadius,day,time);
   dynamicProbabilities = findProbabilities(expectedResources);
   seed = dynamicProbabilities.topHalf()
   agentPath = pickPathUsingRadomization(seed); //to reduce herding effect
   return agentPathWithinRadius()
```



```
def dynamicProbabilities(expectedResources):
   sum = sumOfAllResources(expectedResources);
   // sum is resource sum on leading road segments from junction
   for each road_segment
      probability[road_segment] = expectedResources[road_segment]/sum
```



```
Allocation Framework
def allocationFramework
   for each available resource in Resource stream:
      agent = findClosestAgent(resource)
      assignment[resource] = agent
      resource.state = not_available
      agentList[agent].idle = false
      updateAgentClock()         //Asynchronous Background Task
      updateResourceExpiryClock()     //Asynchronous Background Task
```

6. OUTPUT

- Agent and their search paths.
- Average idle time for agents after one run.
- Average expiration Time of resources for one run.

7. EVALUATION METRICS

- Graph: Idle time against number agents.
- Graph: Distance threshold (for search path algorithm) vs Idle time.
- Graph: Resource expiration time against number of agents
- Graph: Resource expiration time against Distance threshold (for search path algorithm.
- Algorithm performance evaluation against random traversal algorithm.

III.B. INTERMEDIATE ALGORITHM

1. CONSTRAINTS REMAINED THE SAME

- There is no direct or indirect communications between agents.
- Search path of an agent is constrained to the road network and adheres to traffic limitations provided by the map.
- Agent knows neither when nor where the future resources will be introduced.
- No ride sharing allowed. Agent has to empty for assignment.
- Closest (travel time) agent is allocated (always).
- Closest agent's travel time should be less than resource's remaining life time.

2. ASSUMPTIONS

- Every road/path has same speed limit viz 30 miles per hours.
- Every path/road is two way (U turns are allowed).
- Speed limit is stored street wise in the database.
- U turns and one-ways are taken into consideration.
- No influence of traffic conditions, traffic stops etc.
- Herding effect will be reduced/eliminated by random probabilistic selection.

3. OFFLINE DATA STORAGE

Expected Value Table

START LAT	START LONG	END LAT	END LONG	VALUE	TIME	DAY

- Calculate Expected value of number of rides at particular day and time.
- Time is the start time of a five minute period.

Road Segment Table

START LAT	START LONG	END LAT	END LONG	SPEED LIMIT	LENGTH

- This table is filled from data from OpenStreetMap

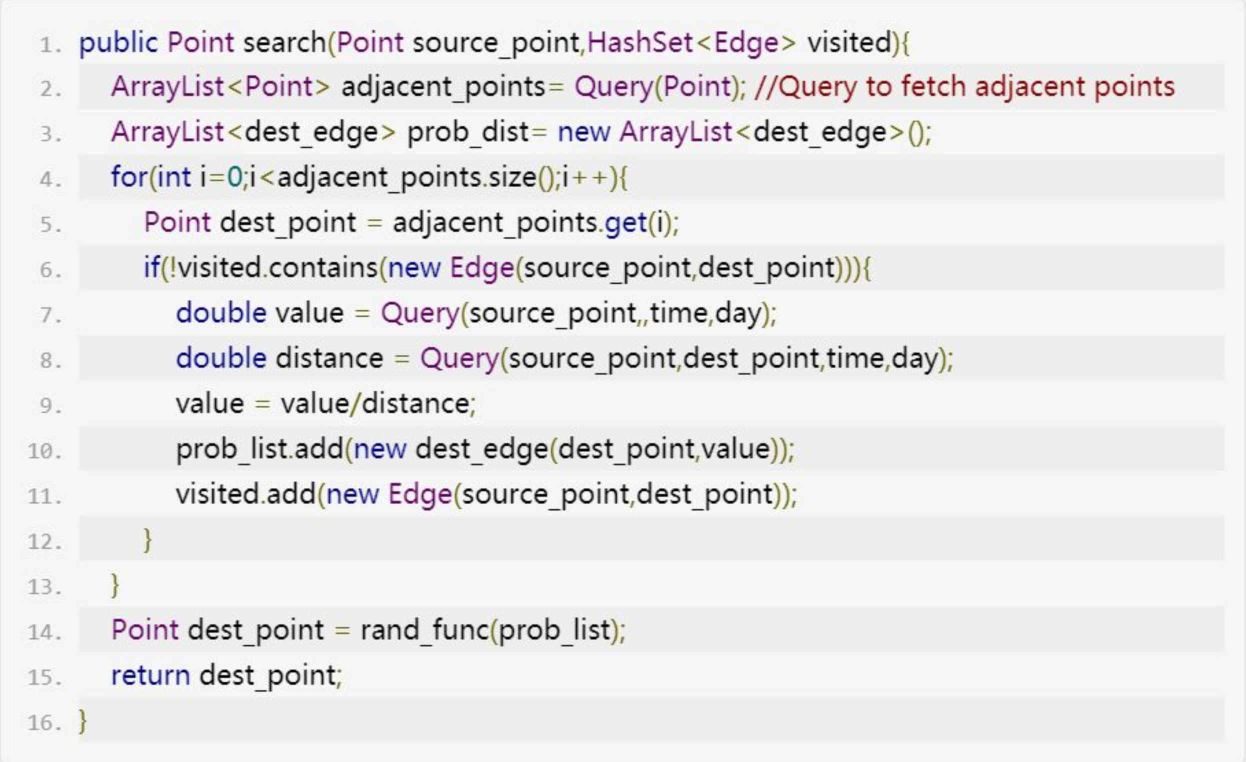
Demand Zones Table

POLYGON	TIME	DAY	LEVEL

- Manhattan is divided into 55 polygons.
- Time is the start time of a five minute period on a particular day.
- A polygon can be categorized into 5 levels based on Resource density.

4. SEARCH PATH ALGORITHM

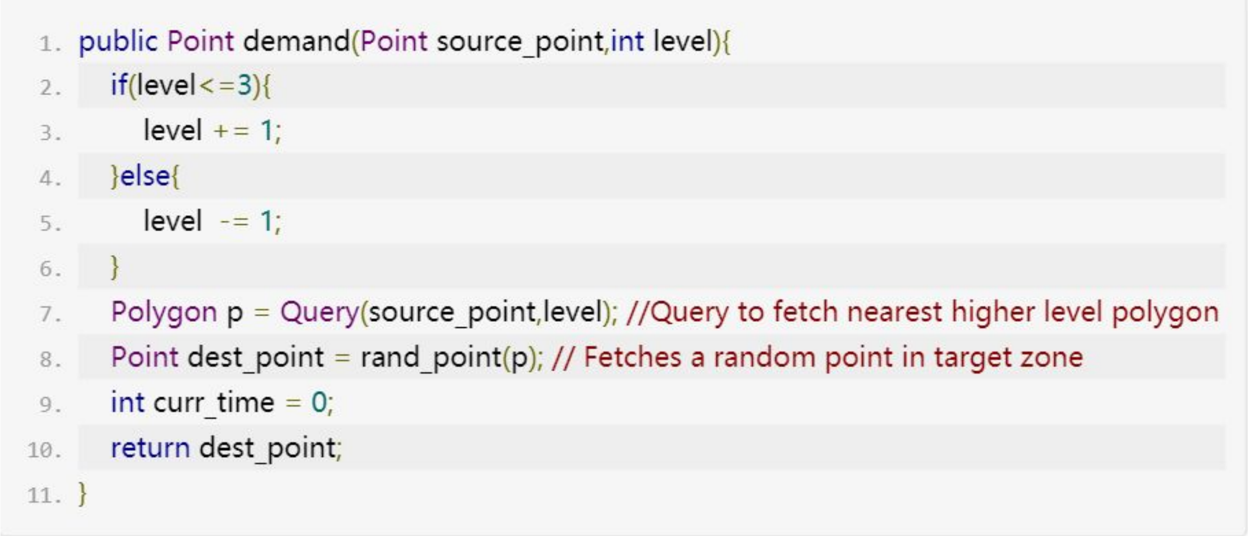
- Retrieve the list of all the edges of the current intersection.
- For each unvisited edge in the list retrieve expected value and edge length.
- Mark each of these edges as visited
- Create a probability distribution using the above retrieved values.
- Perform smoothing if needed.
- Choose an edge at random according to the above distribution.
- If a cab cannot find a resource for 5 minutes then push it to a demand zone.



```
public Point search(Point source_point,HashSet<Edge> visited){
   ArrayList<Point> adjacent_points= Query(Point); //Query to fetch adjacent points
   ArrayList<dest_edge> prob_dist= new ArrayList<dest_edge>();
   for(int i=0;i<adjacent_points.size();i++){
      Point dest_point = adjacent_points.get(i);
      if(!visited.contains(new Edge(source_point,dest_point))){
         double value = Query(source_point,,time,day);
         double distance = Query(source_point,dest_point,time,day);
         value = value/distance;
         prob_list.add(new dest_edge(dest_point,value));
         visited.add(new Edge(source_point,dest_point));
      }
   }
   Point dest_point = rand_func(prob_list);
   return dest_point;
}
```

5. DEMAND ALGORITHM

- If the cab is in a zone whose level is 1-3, then increase its level by 1.
- If the cab is in a zone whose level is 4 or 5, then decrease its level by 1.
- Fetch a polygon nearest to the cab whose level is equal to the updated level.
- Choose a point in this polygon at random and navigate to it.
- Reset the current timer to zero.



```
public Point demand(Point source_point,int level){
    if(level<=3){
        level += 1;
    }else{
        level  -= 1;
    }
    Polygon p = Query(source_point,level); //Query to fetch nearest higher level polygon
    Point dest_point = rand_point(p); // Fetches a random point in target zone
    int curr_time = 0;
    return dest_point;
}
```

6. RESOURCE ALLOCATION ALGORITHM

- Fetch the list of all cabs which are unoccupied by a resource.
- Calculate the time taken for each cab to reach this particular resource.
- If the minimum time to reach that resource is more than its expiration time, allocate the resource to the cab.
- If the minimum time to reach that resource is less than its expiration time, push this resource to a queue.
- Pop out elements from this queue in the next iteration of resource allocation.

7. OUTPUT

- No. of resources expired.
- Average idle time for agents after one run.
- Average resource wait time of resources for one run.

8. EVALUATION METRICS

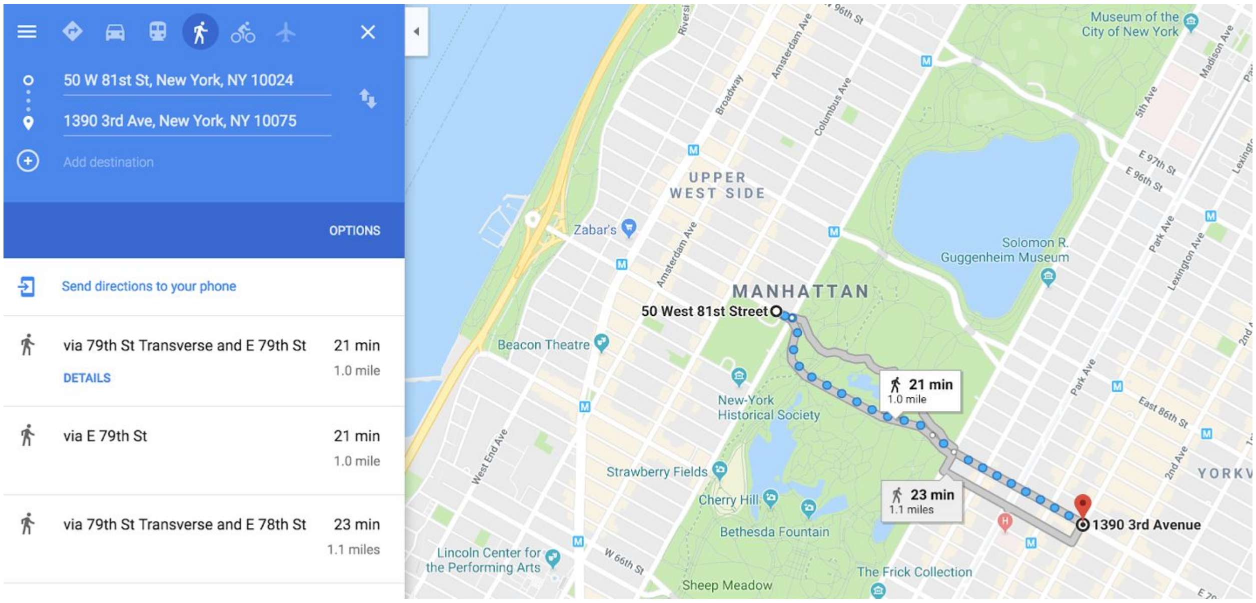
- All plots will compare search algorithm vs random traversal algorithm.
- Graph : Avg Idle time vs No. of cabs.
- Graph : Avg Idle time vs Frequency of resource request.
- Graph : Avg resource wait time vs No. of cabs.
- Graph : No. of resources expired vs No. of cabs

IV. CHANGE OF IDEA

1. FEEDBACK

- **Concerns about 1 Mile Radius:**

It was pointed out to us that there need not be only 15-20 possible paths and that there could be many more in a 1 mile radius. An example for this could be very clearly seen in the Manhattan mile. As the number of blocks significantly increases (in this case close to 20 blocks in the 1 mile radius)- the number of possible paths that can be taken by a cab also significantly increases. We were told to keep in mind - how many possible paths exist given a certain location and its search radius.



50 W 81st St, New York, NY 10024

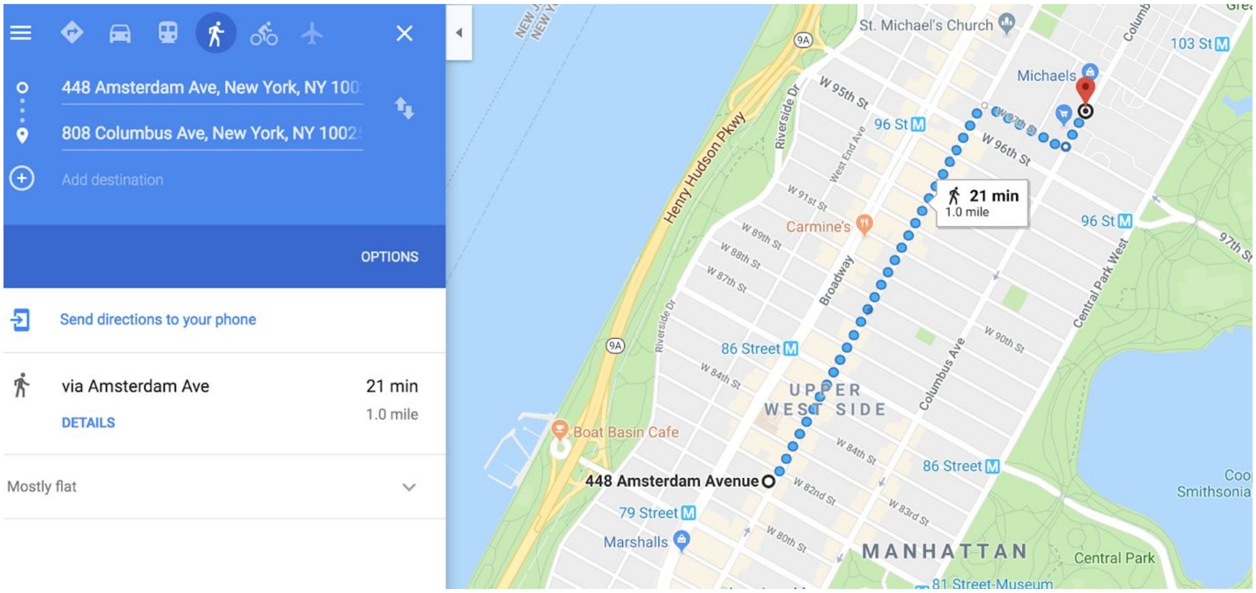
1390 3rd Ave, New York, NY 10075

Add destination

OPTIONS

Send directions to your phone

via 79th St Transverse and E 79th St DETAILS	21 min 1.0 mile
via E 79th St	21 min 1.0 mile
via 79th St Transverse and E 78th St	23 min 1.1 miles



448 Amsterdam Ave, New York, NY 100

808 Columbus Ave, New York, NY 1002

Add destination

OPTIONS

Send directions to your phone

via Amsterdam Ave 21 min

DETAILS 1.0 mile

Mostly flat

- **Expected no of resources Vs Expected time to find a resource:**

  Instead of having expected number of resources as a parameter that your machine learning algorithm will determine, determine the expected time that it will take one to find a resource (on a particular road segment/ 1 mile path/ whatever)

- **Order of task priority:**

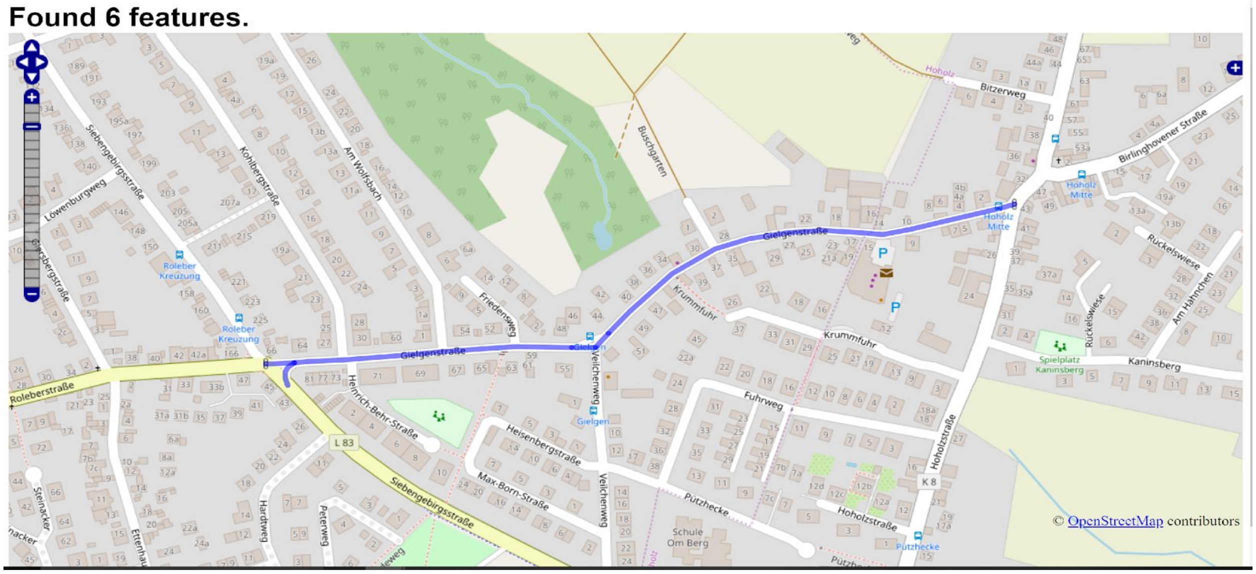
  As for then, we were told to think and focus only on search path algorithm. Later, if COMSET is not provided, we were told to think about simulation

  Search path algorithm > Simulation > Allocation

- We were told to consider **public holidays, weather** data, day, time
- Reinforcement learning

2. OSM DATA WAS PROVING TO BE DIFFICULT

We wanted our map network to represent a graph, where nodes represent intersections and edges represent lanes and singular roads between the closest intersections. But, the OSM data by default - without using API gives data such that a road (is almost intact) and it doesn't break the road into Chunks. It breaks road only at an angle- Ashland (line stream) - all intersections in between will ignored. If we directly load the data, for a road like Ashland, we'd probably get 3 or 4 chunks at points were Ashland deviates from a straight line. As we wanted lanes to represent edges and intersections to represent nodes and such a map is not available by default through OSM. An API called Overpass API should be used on top of OSM, which will take OSM data and it will break it into chunks of intersections of a road like **Ashland**.



Found 6 features.

V. IMPLEMENTATION USING Uber- H3

1. H3: Uber's Hexagonal Hierarchical Spatial Index:

The H3 indexing system is open source and available on GitHub. Grid systems are critical to analyzing large spatial data sets, partitioning areas of the Earth into identifiable grid cells where each grid is a hexagon. The basic functions of the H3 library are for indexing locations, which transforms latitude and longitude pairs to a 64-bit H3 index, identifying a grid cell. The function *geoToH3* takes a latitude, longitude, and resolution (between 0 and 15, with 0 being coarsest and 15 being finest), and returns an index. *h3ToGeo* and *h3ToGeoBoundary* are the inverse of this function, providing the center coordinates and outline of the grid cell specified by the H3 index, respectively.

Once you have data indexed using H3, the H3 API has functions for working with indexes to get neighbors at different levels referred as k-rings, center address etc. We divide Manhattan – which is the area under consideration into 516 grids, each identified by a hexagon ID, which

2. CONSTRAINTS REMAINED THE SAME

- There is no direct or indirect communications between agents.
- Search path of an agent is constrained to the road network and adheres to traffic limitations provided by the map.
- Agent knows neither when nor where the future resources will be introduced.
- No ride sharing allowed. Agent has to empty for assignment.
- Closest (travel time) agent is allocated (always).
- Closest agent's travel time should be less than resource's remaining life time.

3. DATA PREPROCESSING

- In Python, import H3
- Download Yellow cab data and read it
- For each ride data's point

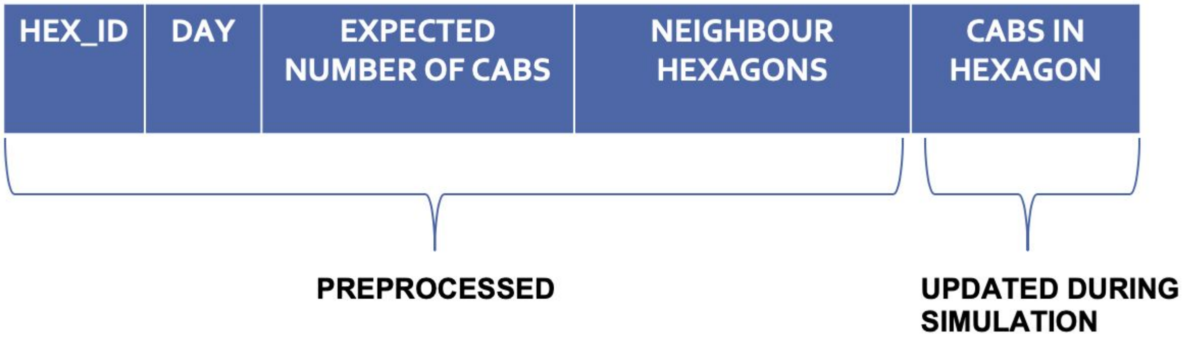
  - **find Hex ID of pickup location** by converting it to a Hexagon object, referenced by its object index and then attaching it as a column for the same data
  - **add day of the week column** by rounding off the pickup time to the nearest five-minute window
- Create a data set which takes 4 points as boundary of Manhattan area and polyfill it with hexagons. It will return all hexagons created in it.
- You’ll get 524 hexagons. Center, Find k-rings for that hex id and add it to the hexagon table referenced by hexagon ID’s.
- Aggregate the yellow cab data to get the resource count in a hexagon for a particular day of the week, for a particular time interval. Add this value as the expected number of resources to the Hexagon Table such that the Hexagon table is referenced by HexagonID, Time Interval, Day of the Week
- Loaded both the data into DB,

4. OFFLINE STORAGE - (Preprocessed Data)

- tpep_pickup,
- tpep_dropoff,
- passenger_count,
- trip_distance,
- pickup_longitude,
- pickup_latitude,
- dropoff_longitude,
- dropoff_latitude,
- hexagon_id,
- day_of_week

5. DATA TABLES

Hexagon Map



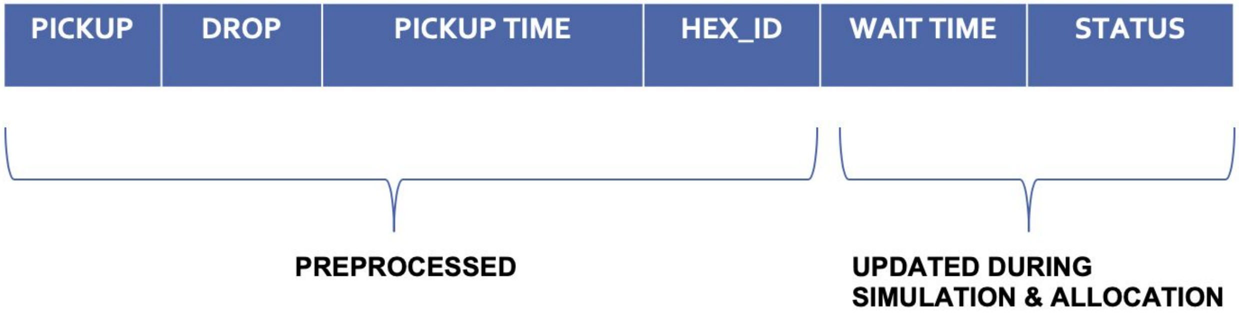
HEX_ID	DAY	EXPECTED NUMBER OF CABS	NEIGHBOUR HEXAGONS	CABS IN HEXAGON

PREPROCESSED

UPDATED DURING SIMULATION

- Calculate Expected value of number of rides at particular day and time.
- Time is the start time of a five-minute period.
- As each cab moves to a new hexagon, update hexagons to reflect the cabs it contains

Resource Queue



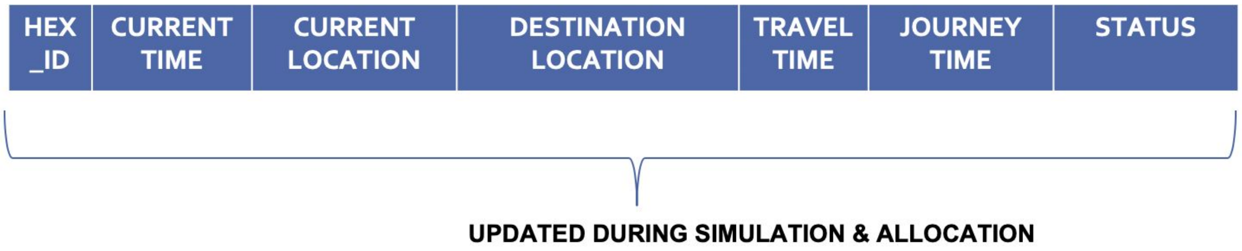
PICKUP	DROP	PICKUP TIME	HEX_ID	WAIT TIME	STATUS

PREPROCESSED

UPDATED DURING SIMULATION & ALLOCATION

- Calculate wait time associated with each resource before it is allocated or before it achieves MLT

Cab List

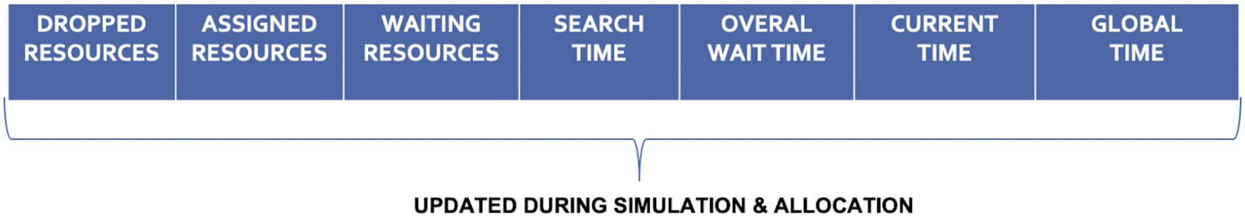


HEX _ID	CURRENT TIME	CURRENT LOCATION	DESTINATION LOCATION	TRAVEL TIME	JOURNEY TIME	STATUS

UPDATED DURING SIMULATION & ALLOCATION

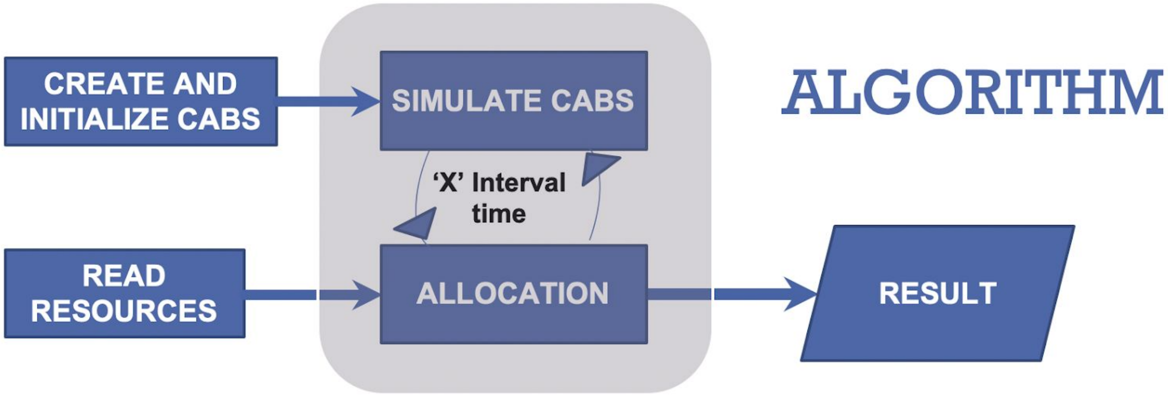
- Cabs created and initialized at equidistant hexagons in map
- Travel time – from the time resource has been allocated to pickup of resource and until drop off
- Journey time – time from cab location to pick-up and from pickup to drop off

Result



- Search time - search time- time taken from beginning to when the cab found a resource
- Wait time- time from resource's appearance to when it has been allocated

4. ALGORITHM- BIG PICTURE



- Obtain Resource list - given CSV of rides between input startTime and endTime
- Create and initialize cabs
- For the given duration between start and end time
    1. Simulate cab movement in each iteration
    2. Allocate the cabs to the resources
    3. Updated the global time by '**x**' at the end of the iteration

5. SIMULATION - simulates cab movement

For each cab,

- Get neighbor hexagons of each cab’s current hexagon and their expected values
- Using a random probability distribution, pick a neighbor hexagon
- Set this hexagon as the destination hexagon that the cab will move to
- Find the time taken to move from the current hexagon to destination hexagon
- Decrement this travel time and at zero time (when cab reaches destination hexagon) in one of next iterations of simulation,
    1. add this cab to the new destination hexagon
    2. remove the cab from the old hexagon

6. ALLOCATION – allocates resources

In each iteration, we process resources one by one

- Get (k ring) neighbor hexagons of each resource’s hexagon and run through all their cab lists (we compare at most 60 neighbor hexagons for any resource)
- Find cab with min distance from resource
- Allocate the closest cab to the resource if the time taken to pickup is less than current MLT (initial MLT – already expired wait_time of resource)
- Otherwise add the resource back to resource queue and increment wait time for resource
- If waiting time of the resource > MLT then increment total_dropped

VI. EQUATIONS USED

Time jumps

current_time = current_time + time_window

Weighted Random

First Ring of neighbors contains at most 6 neighbors. So, size of all the following collections is size of neighbors in first Ring **= firstRingSize**

firstRing contains indices of all the neighbors from, **neighbors** are the object ID's of all first ring neighbors from, **expected** contains all the expected number of resources for these neighbors

prefix_i	$\text{expected}_{\text{neighbor}(i-1)} + \text{expected}_{\text{neighbor}(i)}$ **, i = [0, firstRingSize]**
r	$\text{random} * \text{prefix}_{(\text{firstRingSize}-1)} + \text{prefix}_0$

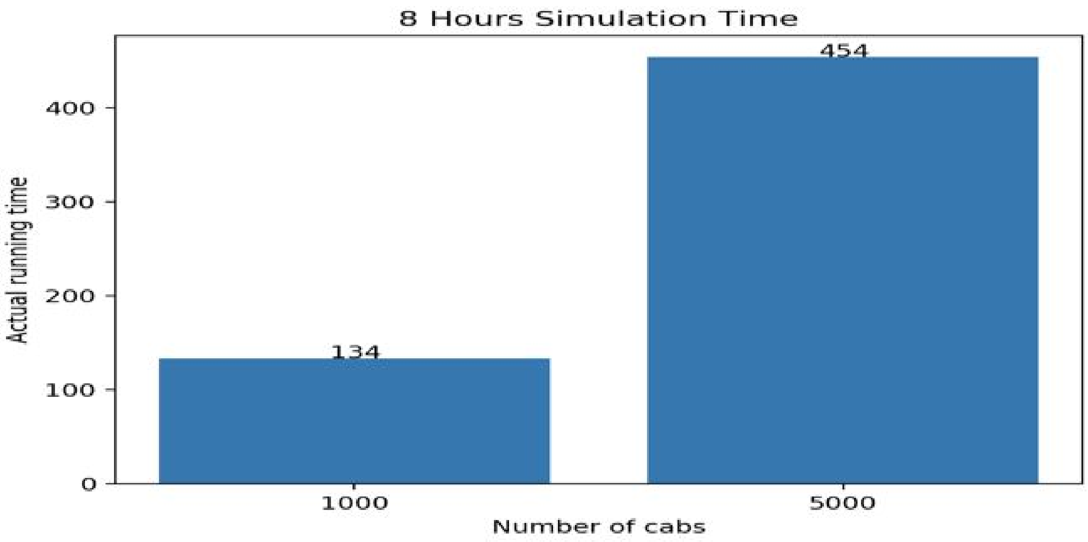
VII. HOW EXPERIMENTS WERE CONDUCTED?

1. Getting the closest cab by comparing the **distance** between a resource and all available cabs**.**
2. (Based on feedback) Getting the closest cab by comparing the **distance between a resource and the cabs present only in its k-nearest neighbors** (k- rings).
3. By **changing the time window** of iteration each time i.e each looping iteration jumps time based on the time window specified. For example, a time window of let's say 1 minute would mark each successive iteration to refer to a minute passed in simulation time of the test day.
4. Our algorithm uses a weighted probability distribution to return the index of the neighbor. We also run the experiments on an **ordinary random function** such that it picks a number within a range, where the range is [0, total number of neighbors-1]. This number gives us the index of the neighbor that the cab can now traverse to.

SIMULATION	5 MIN	30 MIN	1 HOUR
TIME WINDOWS	1 min, 30 sec, 15 sec	1 min, 30 sec, 15 sec	1 min, 30 sec, 15 sec

VIII. RESULTS AND PLOTS

1. SIMULATION TIME vs RUN TIME



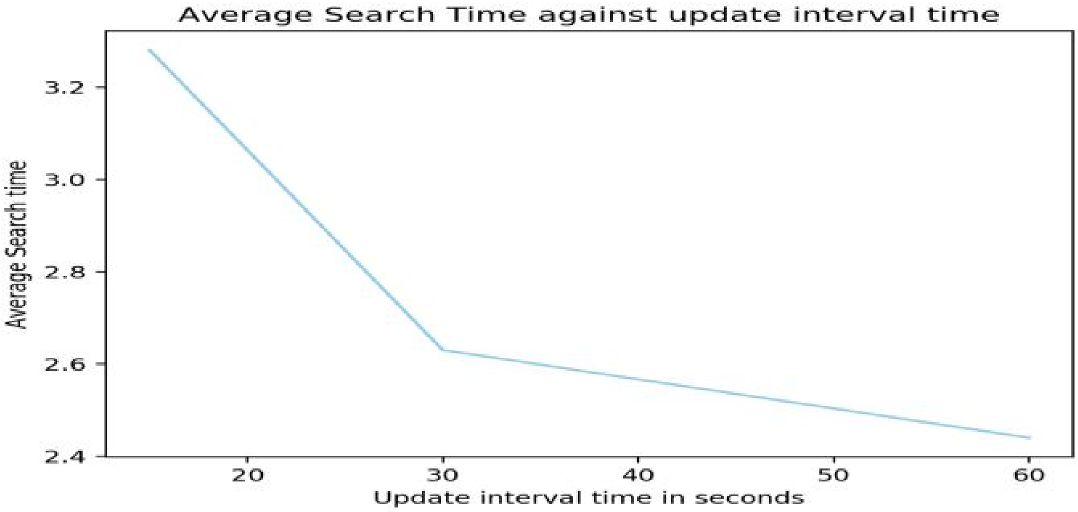
Plot Explanation: The graph depicts the time taken for the algorithm to simulate 8 hours of data.

Results: It took 134 minutes to simulate 1000 cabs for 8 hrs of data.

Justification of Results: The most expensive computation is the request made to the graphhopper to calculate the distance from one point to another while calculating the search path of the agent. A request is made to the graphhopper to calculate the nearest agents from the resource while allocating the cab.

Expectation: Result is as expected

2. AVERAGE SEARCH TIME vs UPDATE INTERVAL



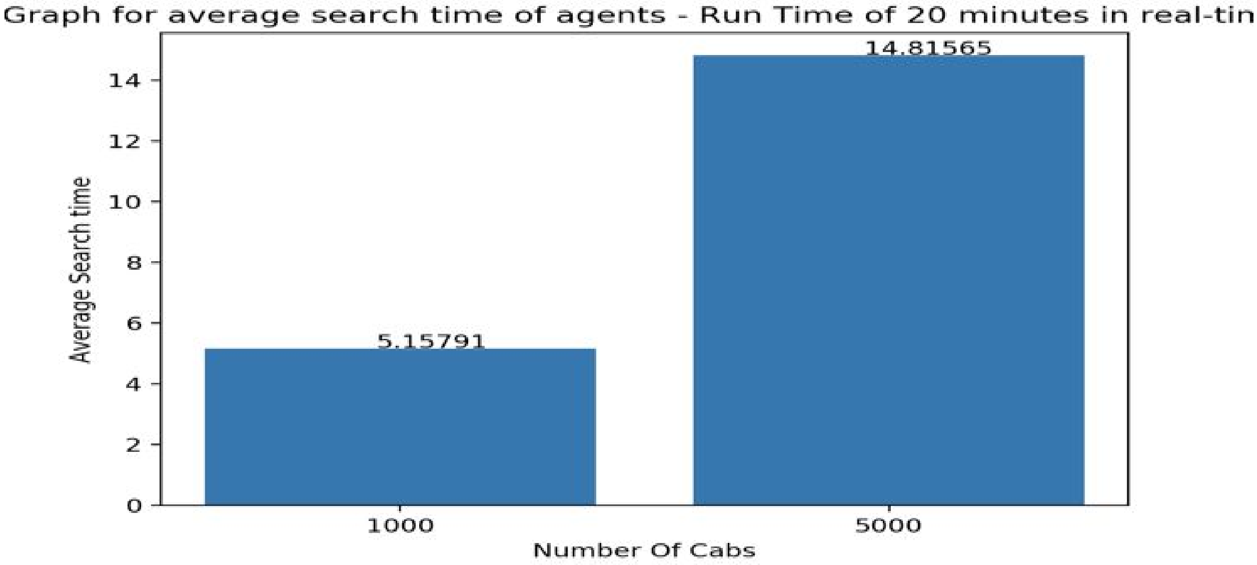
Plot Explanation: This graph is used to evaluate an optimal update frequency for cab simulation and resource allocation. we conducted experiments for each window for a period of 1 hr simulation to find out the best update interval.

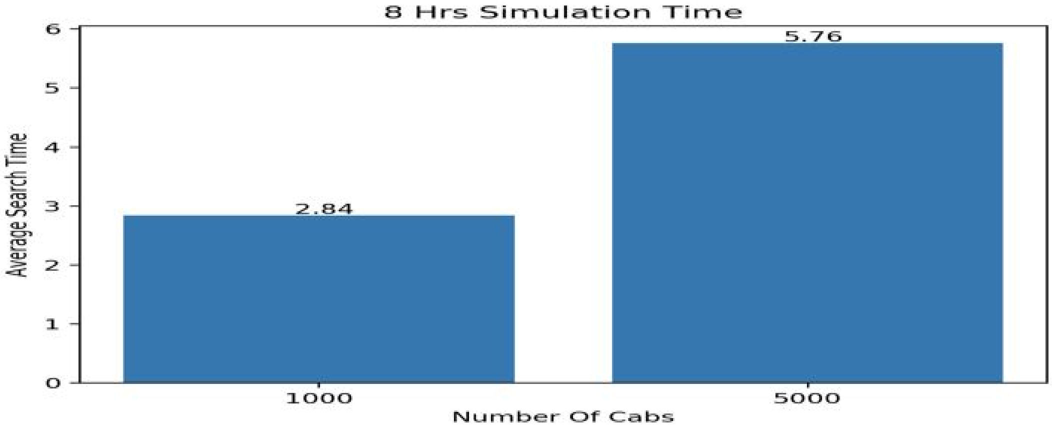
Results: Initially, as time window(update interval) is small, average search time is high and as time window increases, search time decreases. 30 seconds time windows seemed to be ideal.

Justification of Results: The reason for the decline in the curve is due to the granularity. This granularity is inversely proportional to the computation time of the algorithm

Expectation : The results obtained are as expected.

3. AVERAGE SEARCH TIME





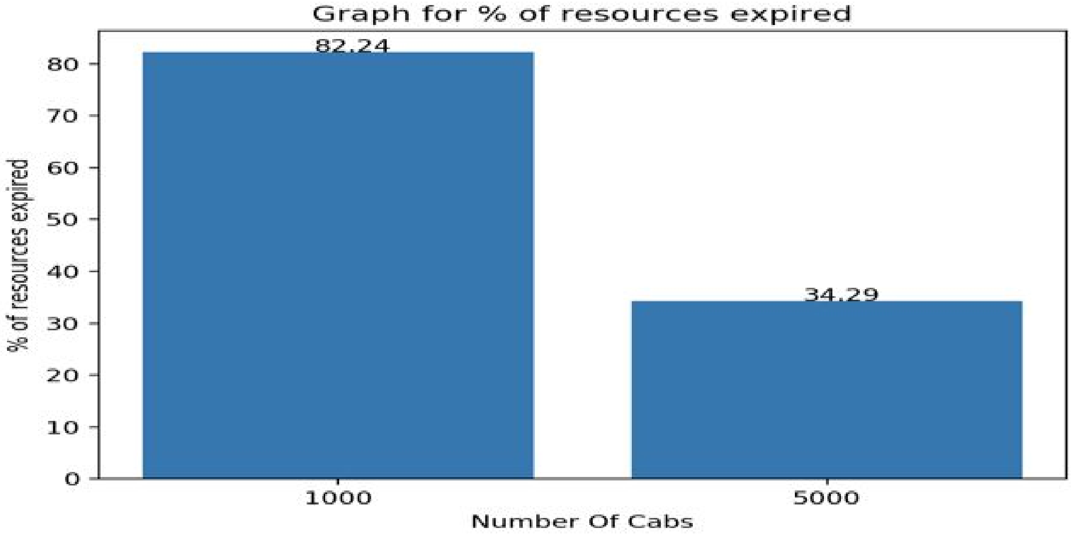
Plot Explanation: The graph depicts the average search time taken by each cab after 8 hr simulation of data.

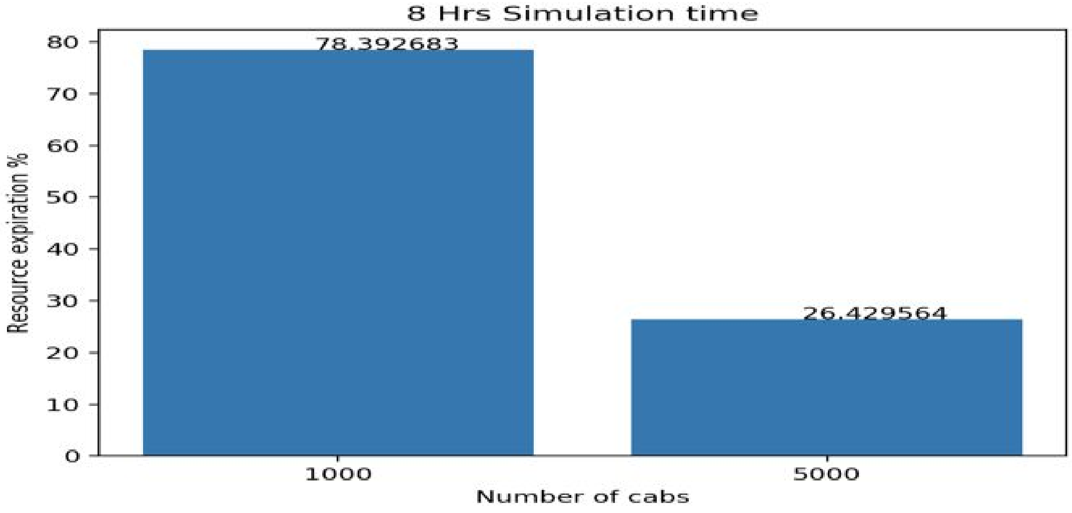
Results: While simulating a thousand cabs, it took about 2.84 minutes while simulating for 5000 cabs it took 5.76 minutes.

Justification of Results: The search time will increase with the number of cabs because some of the cabs are sufficient to serve the current requests while other cabs keep searching until more resources come up. Moreover, the allocation of resource is only done with the nearby cabs (4 rings)

Expectation : The results obtained are as expected.

4. RESOURCE ASSIGNMENT





Plot Explanation: The graph depicts the total resource expiration rate during the simulation i.e how many resources have expired due to the MLT restriction.

Results: For 1000 cabs, it is ~78% and for 5000 cabs, it is ~26%.

Justification of Results: In the case of 1000 cabs, the number of cabs does not meet the demand of resources and ultimately resources get expired. In the case of 5000 cabs, it is more probable that cabs were able to meet the demand and thus the expiration rate is also less.

Expectation : The results obtained are as expected.

IX. SOFTWARE TECHNOLOGY

1. **IntelliJ** - IntelliJ IDEA is a Java integrated development environment for developing computer software. It is developed by JetBrains, and is available as an Apache 2 Licensed community edition.

2. **QGIS** - QGIS is a free and open-source cross-platform desktop geographic information system application that supports viewing, editing, and analysis of geospatial data.

3. **Python** - Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace.

4. **Graphhopper** - GraphHopper is an open-source routing library and server written in Java and provides a web interface called GraphHopper Maps. As well as a routing API over HTTP. It runs on the server, desktop, Android, iOS or Raspberry Pi.

5. **Java** - Java is a general-purpose programming language that is class-based, object-oriented, and designed to have as few implementation dependencies as possible.

6. **Apache Maven** Build Tools - Maven is a build automation tool used primarily for Java projects. Maven addresses two aspects of building software: first, it describes how software is built, and second, it describes its dependencies

X. THIRD PARTY LIBRARIES

1. **PostGIS** - PostGIS is an open source software program that adds support for geographic objects to the PostgreSQL object-relational database. PostGIS follows the Simple Features for SQL specification from the Open Geospatial Consortium. Technically PostGIS was implemented as a PostgreSQL external extension.

2. **Uber H3** - PostGIS is an open source software program that adds support for geographic objects to the PostgreSQL object-relational database. PostGIS follows the Simple Features for SQL specification from the Open Geospatial Consortium. Technically PostGIS was implemented as a PostgreSQL external extension.

3. **PostgreSQL** - PostGIS is an open source software program that adds support for geographic objects to the PostgreSQL object-relational database. PostGIS follows the Simple Features for SQL specification from the Open Geospatial Consortium. Technically PostGIS was implemented as a PostgreSQL external extension.

XI. COLLABORATION TOOLS

1. Google Docs
2. Git
3. Source Tree
4. WhatsApp

XII. REFERENCES:

- Yellow Taxi Trip Data:

  https://data.cityofnewyork.us/Transportation/2016-Yellow-Taxi-Trip-Data/k67s-dv2t
- OpenStreetMap data to PostgreSQL converter

  https://github.com/openstreetmap/osm2pgsql
- Overpass API

  https://wiki.openstreetmap.org/wiki/Overpass_API
- Loading OSM Data with Overpass API

  https://towardsdatascience.com/loading-data-from-openstreetmap-with-python-and-the-overpass-api-513882a27fd0
- Using QuickOSM to download OpenStreetMap data in QGIS

  -http://jonathansoma.com/lede/foundations-2018/qgis/osm/
- Loading the OSM Planet data into PostGIS

  https://www.bostongis.com/PrinterFriendly.aspx?content_name=loading_osm_postgis
- Github- Uber H3

  https://github.com/uber/h3-py
- Random number generator in arbitrary probability distribution fashion

  https://www.geeksforgeeks.org/random-number-generator-in-arbitrary-probability-distribution-fashion/