CS 581

RESOURCE SEARCH - TEAM 9

Akshun Jhingan

Amrish Ashvinkumar Jhaveri

Chinmay Gangal

Mohitkumar Paritosh Ghia

Table of Contents

Introduction

This project focuses on competitive spatiotemporal search (CSTS), in which, mobile agents search for stationary resources in a road network. Each resource can be obtained by only one agent at a time. Each agent autonomously decides its own search path in an attempt to obtain a resource as quickly as possible; therefore, the search is competitive. A search problem of this nature arises in applications that are very commonplace in typical urban transportation systems. Some of them are as follows:

- Crowdsourced taxicabs (mobile agents) looking for customers (stationary resources).
- Vehicles (mobile agents) looking for available parking slots (stationary resources).
- Electric cars (mobile agents) looking for available charging stations (stationary resources)

A key issue to be addressed in the CSTS problem is how agents should choose their search path such that the search time is minimized. We propose a framework in which we try to solve the competitive spatial-temporal search for crowdsourced taxis where the location of resources is not known to the drivers in advance. We assume that there is a historical dataset containing records of the times and locations at which resources became available in the past, such as TLC for cab customer availability. From such datasets, the distribution of resources may be derived and utilized by the agents to plan their search paths. For example, the distribution can define the probability that there is at least one resource available during each time interval for each spatial zone.

However, what is the right granularity to use for the time interval and spatial zone remains an open question. Furthermore, because all the agents plan their search paths based on the same resource distribution information, they tend to choose common paths. This causes the “herding” effect, which can hurt the search efficiency.

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 fashion, each with a destination. Each resource has a maximum lifetime (MLT) starting from its introduction, beyond which the resource will be automatically removed from the system, an outcome which we will call resource expiration. 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 lifetime (i.e., MLT minus the time duration since the resource is introduced until the present time).

Formal Problem Definition

Input

A road network (map), A training dataset (list of resource locations and timestamps), test dataset (list of resource locations and timestamps), The number of agents (i.e., agent cardinality) and their initial locations, the agent cardinality 1000 and 5000

Output

Search path for each agent.

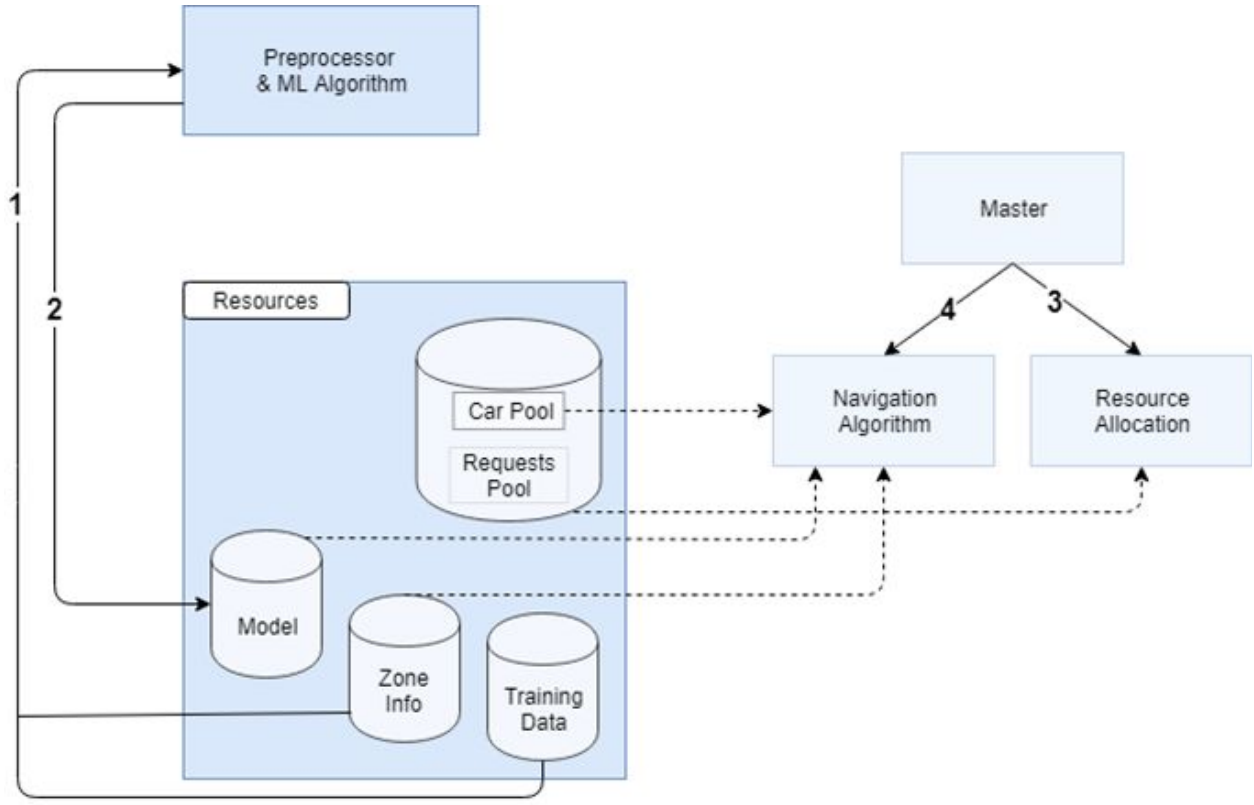
Objective

Minimizing (in order of priority) the average search time, average wait time, and expiration percentage. The results are shown by graphs.

Assumptions and Restrictions

Area covered: Manhattan, Car Speed: 15 mph, Data: Jan - June 2016, A car location is at Centroid of Zone, All Zones are connected logically, Traffic conditions are not taken into consideration, cabs do not interact with each other

System Architecture



The system functions as follows:

1) Zone Information and TLC records are fed to our preprocessing component, which discards all non-Manhattan records and erroneous records. Following that the additional information like season and day type are added to each record.
2) These processed records are then fed to our machine learning model for training and then model gets dumped in local memory for its use in API.
3) The system before starting makes the request to API to get the zones scores for the navigation component. These scores get refreshed in every 30 minutes so that the system performs well according to history.

4) Now as the test data is provided, it is pre-processed by our system for removing erroneous records and provided to our simulator.
5) The simulator reads the records of resources in every 50 secs and appends them to the resource pool.
6) Our cab resource allocation component gets called to find a match for available resources sorted according to their expiration time.
7) Finally, the navigation component is called over empty cabs with no assigned path to a path of k-hops.

Pre-processing

Manhattan is divided into 546 hexagonal zones using Uber H3 API at resolution 9. The distance between two neighboring centroids = 0.34 km i.e. 0.21 mi. Using Random Sampling, we took 20% of records from January to June 2016.

We consider these columns from the original dataset: Trip Id, Pickup Latitude, Pickup Longitude, Pick up timestamp, Drop Off Latitude, Drop Off Longitude, Drop off timestamp, Trip Distance. We add new data like zone id (from H3 API) for pickup and drop off locations, season - Fall, Summer, Spring, time slot (1- 48 for a single day), day type - weekday or weekend for each record.

Following are a few types of trips which are filtered out from the dataset:

- Trips with the same pickup and drop off time - these are canceled trips
- Trips neither originating nor ending in Manhattan

We trained our ML model on this training data.

Algorithms Evaluated

Machine Learning Models

1. Linear Regression

Linear Regression is a supervised machine learning algorithm where the predicted output is continuous and has a constant slope. It attempts to model the relationship between two variables by fitting a linear equation to observed data. In our algorithm, it is used to predict count and average trip duration.

The variables Season, Day type, Timeslot, Zone represent the attributes or distinct pieces of information, we have about each observation.

We predict the count and average trip duration based on the features mentioned above. We then use the weighted average of both these predicted scores.

Final_score = (w1 * count) + (w2 * avg_trip_time)

2. Polynomial Features

The same linear regression model is combined with polynomial degree features.

PolynomialFeatures - The data resembles polynomial curves more closely than a linear line. Therefore, to overcome under-fitting by linear line, we need to increase the complexity of the model. Polynomial features module converts the original features into their higher

order terms. We tried polynomial features but due to lack of resources, we are not able to train the model on more than 6000 records.

Navigation

Starting from the cab's current zone, the algorithm calculates the next path of k-hops by non-deterministically selecting from neighbor zones based on their individual scores. Setting a higher 'k' reduces the number of cabs to manage navigation for once a destination and estimated available time is known to the server. This algorithm runs for each unassigned cabs from the pool of cabs

Experimentation

H3 Zones

We first require all the H3 Zones inside Manhattan at resolution level 9. By providing geo-JSON coordinates of the polygon formed around Manhattan to the H3 API's *polyfill*, we get all the hexagonal zones within this polygon. Now for each such hexagonal zone, we require to find its neighbors based on the size of the ring (k). We maintain a list of neighbors for each value of k i.e. k=0 would contain the same zone, k=1 would contain all neighbors adjoining the current zone and likewise for k=2. For each neighboring zone, we also check if it is within Manhattan. We generate a JSON file containing a map(key-value pairs) of the zone id (h3 index) and the zone details about its neighbors, latitude, and longitude.

Simulation

As described before, we also have Python Flask API which provides the zone scores for that time-slot. This API is hit every 30 minutes of the simulation time.

We create a JAR (Java Archive) of the simulation code and deploy it on the AWS EC2 instance where we execute it with different parameters.

Read pre-processed CSV

The CSV for the resources is pre-processed so we have only the correct records. Also, pick up and drop off location in latitude and longitude are converted to their h3 index using the Uber H3 API.

While reading the CSV and instantiating the Resource object, we also provide the expiration time left for the resource - initialized to 10 minutes. Since the resource request is not sorted based on the pickup time in the original dataset, we sort the dataset based on pickup time.

Simulation Time

The simulation time starts from the earliest pickup time. Each iteration will increment the simulation clock by 50 seconds - calculated from H3 distance between 2 centroids and cab's constant speed of 15 mph.

Every 30 mins of simulation time we get the latest zone scores from the Flask API providing the current simulation time to it.

Resource Allocation

Before we can assign nearest cabs to the resource, we need to do the following for each iteration of the simulation:

- Find available/free cabs: From the cab pool, we find the free cabs i.e. currently no resource is assigned or if assigned, then the next available time is passed from the current simulation time.

- Current Pool of resources: Iterating over all the resources (from the previous break-point), we find the next resources to be considered in the current simulation iteration. We also consider any resources not assigned from the previous iteration if their expiration time is greater than 0. We sort these resources based on their expiration time i.e. resource with least expiration is given priority.

Once we have a free cabs pool and current pool resources, we find the nearest cab for each resource. When we assign a cab to the resource, we remove it from the cabs pool so the cab is not considered for the next resource in the same iteration.

Navigation

In each simulation iteration, for each available cab (after the above resource allocation is done), we find the next zone it should navigate to. This navigation runs in parallel to decrease the running time of the simulation. The navigation logic was described above.

End of Simulation

The simulation end if any of the constraints are met:

- No more resources are left to be considered i.e. all resources in the csv are considered and resources have either been assigned or have expired.
- If the target running-time to end the simulation has been reached. We provide a provision to configure the simulation to run for *k* minutes e.g. 10 minutes.

Metrics Calculation

Since we require a variety of metrics at the end of the simulation, we have created various utilities to generate the results. We provide the cab pool, assigned resources, expired

resources and the current pool of resources that were considered in the last simulation iteration.

Save to JSON file

Once the simulation is completed, we write the cab pool, assigned resources, expired resources to JSON files.

Commands to Run the Jar

First, we need to build the jar from the source code:

Assuming Apache Maven 3 with Java 8 is installed on the system, open the *AllocationNavigation* folder and run the following command from the terminal:

mvn clean package

Now we need to transfer this jar to AWS EC2 instance either using WinSCP (for Windows) or scp from Ubuntu / OS X.

We also need to copy the required resource files to the same location. The files required are:

- havershine.json : Contains precomputed distance between all H3 zones inside Manhattan. This speeds up the simulation considerably.
- manhattan_zones_lat_lon_3.json: The Zone information required in the form of a map (key, value pairs).
- preprocessed.csv: The preprocessed CSV which we will provide to the simulation.

To execute the JAR, we run the following command:

nohup java -Xmx40G -jar resource-search-1.0-SNAPSHOT-jar-with-dependencies.jar 1000 25 5 600000 0 5 false false false &

nohup - since we want our process to be detached from the current shell session

-Xmx40G - We make 40 GB of RAM available for the JVM to execute this jar.

We provide 9 commandline arguments:

1. Number of cabs to consider.
2. Speed of the cab in kph (kilometer per hour)
3. Number minutes to run the simulation
4. The expiration time for each resource in minutes
5. Number of minutes before the pickup time, which would be the request time.
6. The value of *k*, used for navigation. We calculate *k* hops in advance but navigate only 1 hop i.e. 1 zone in each simulation iteration for navigation.
7. Boolean flag for reading zone score from the file. Used while doing dry runs. It should be set to false.
8. Boolean flag to consider all the records of the CSV. If set to true the parameter 3 is no longer considered.
9. Boolean flag to generate random scores for the zone. Used while doing dry runs. It should be set to false.

So we set the values for each experiment accordingly and wait for the execution to complete.

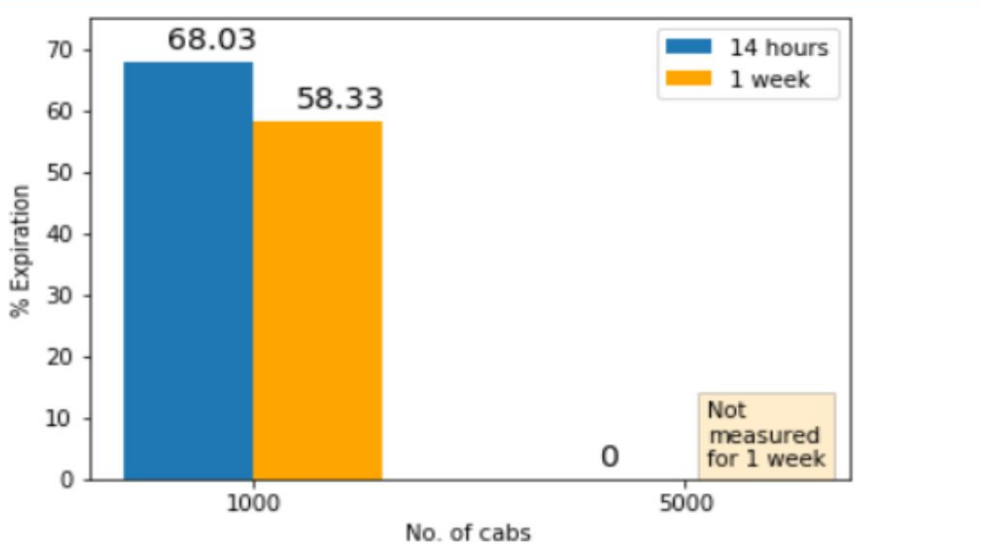
Results and Plots

Logs from a week's run with 1000 cabs for May 2016

- Duration: May 1 00:00:00 2016 - May 7 23:59:59 2016
- Simulated for: 10080.191716666666 mins
- System ran for: 593.4841 mins
- Average Search Time per cab per resource: 6.334950666666666 mins
- Average Idle Time per cab per resource: 3.9073389833333336 mins
- Average resources picked up per cab:978.529
- Expired Resource count: 1369626
- % Expired Resources: 58.33016049859075
- Assigned Resource count: 978432
- % Assigned Resources:41.66983950140925
- Total resources did not arrive yet: 7330717
- Total resources in the csv: 9680953

Note: It is very costly to simulate a week for 5000 cabs, hence it was not measured.

Average Expiration (%)

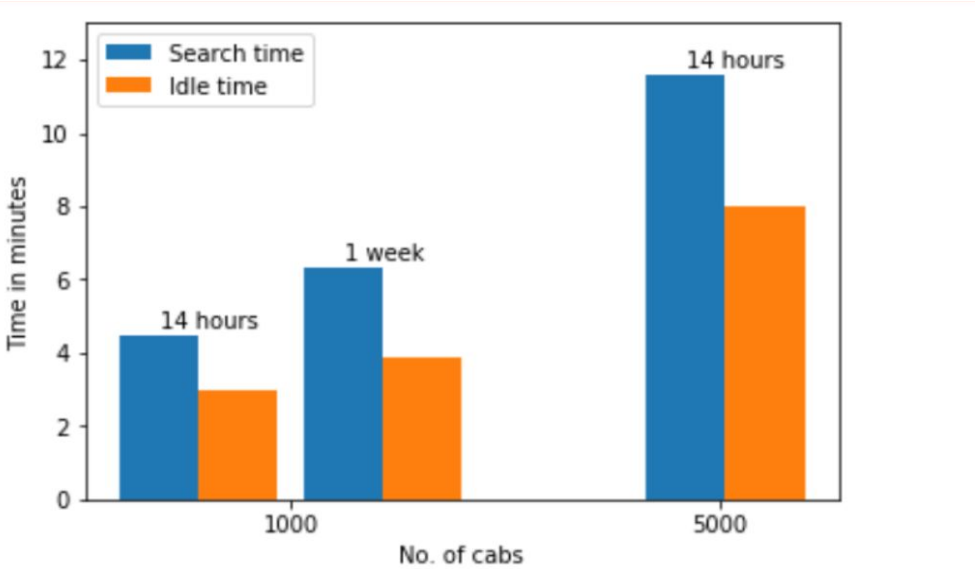


This plot shows the avg expiration percentage by comparing the number of cabs and the total time of the simulation. Some insights from this graph are:

a) For 1000 cabs, 1 week of simulation time gives lower expiration percentage as compared to 14 hours simulation because a week comprises of both day and night causing more chance of serving resources because of less demand during the night.

b) When 5000 cabs are there in the system the expiration percentage is 0 since more cabs are able to serve the resource as compared to 1000 cabs.

Search / Idle Time



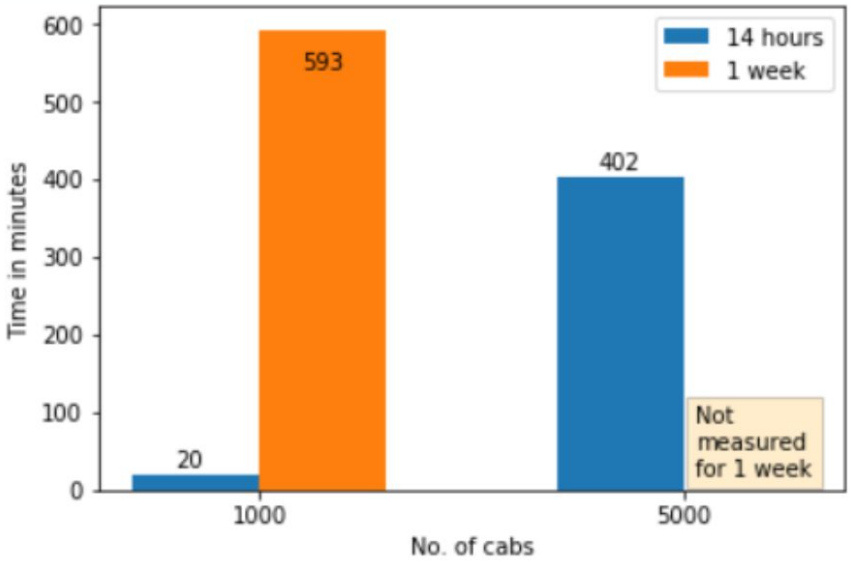
Some definitions:

1) Idle time - Time from the last drop off to the time cab is assigned a resource
2) Search time - Time between the drop-off and the next pickup

This plot shows the avg idle/search time by comparing the number of cabs and the total time of the simulation. Some insights from this graph are:

a) For 1000 cabs, 1 week of simulation gives both higher idle and search time as compared to 14 hours simulation because a week comprises of both day and night causing more empty cabs because of fewer resources during the night.
b) For 5000 cabs, the idle and search time is high since more cabs are available compared to resources during 14 hours of simulation.

Running Time Comparison

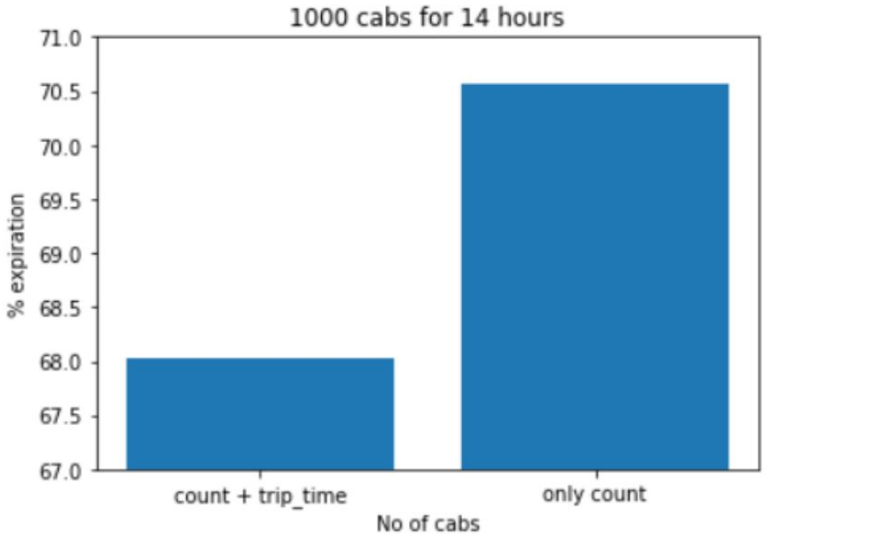


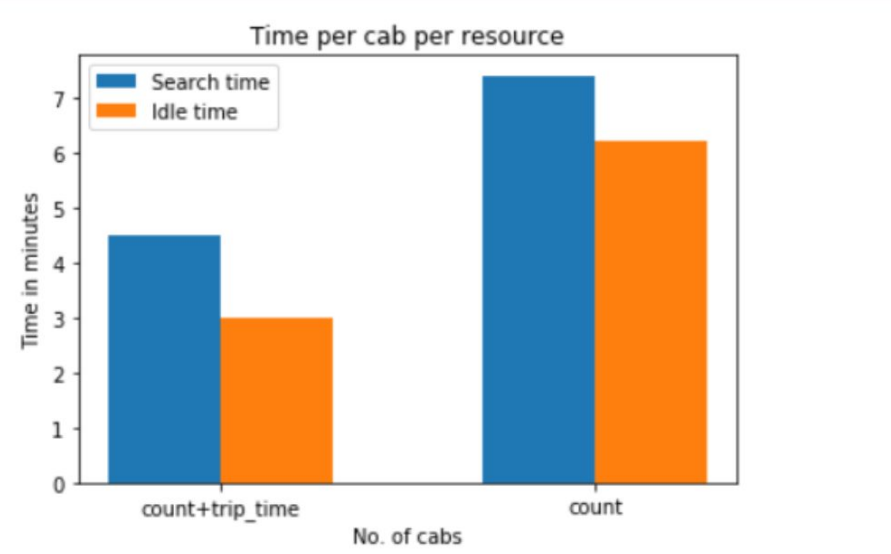
This plot shows the running time of our system by comparing the number of cabs and the total time of the simulation. Some insights from this graph are:

a) Since we had highly parallelized code we were able to simulate the system in considerably less time.

b) For 5000 cabs the running time is high naturally since more empty cabs make requests for navigation.

ML Model Comparison



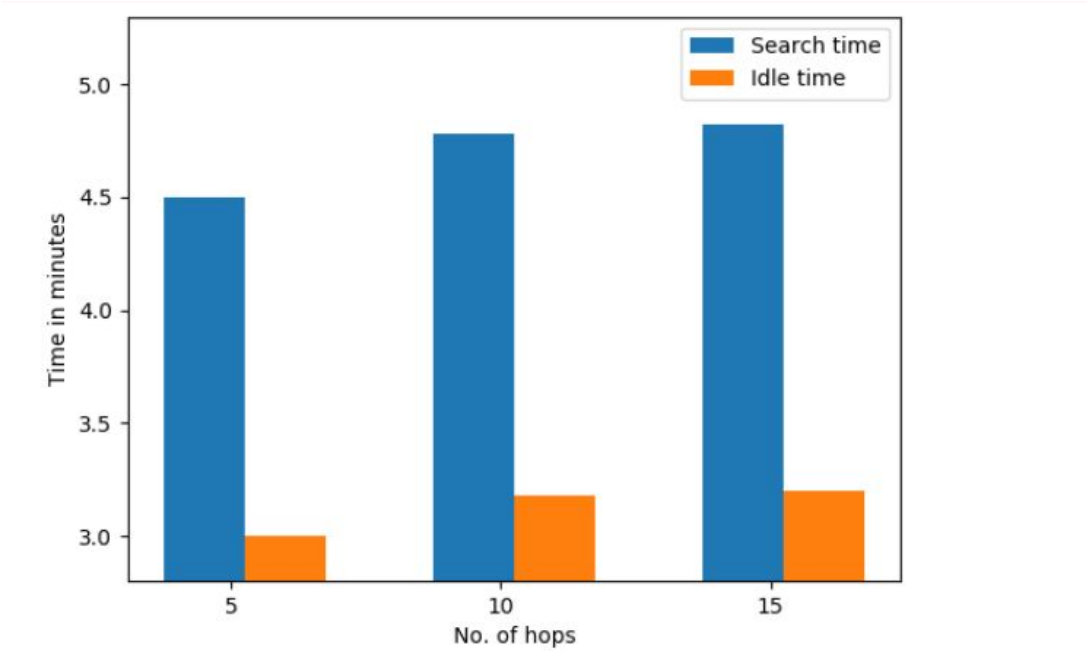


We tried two models for scoring a zone. This plot shows the comparison between them in terms of the average expiration percentage they were able to save. Some insights into choosing the weighted model:

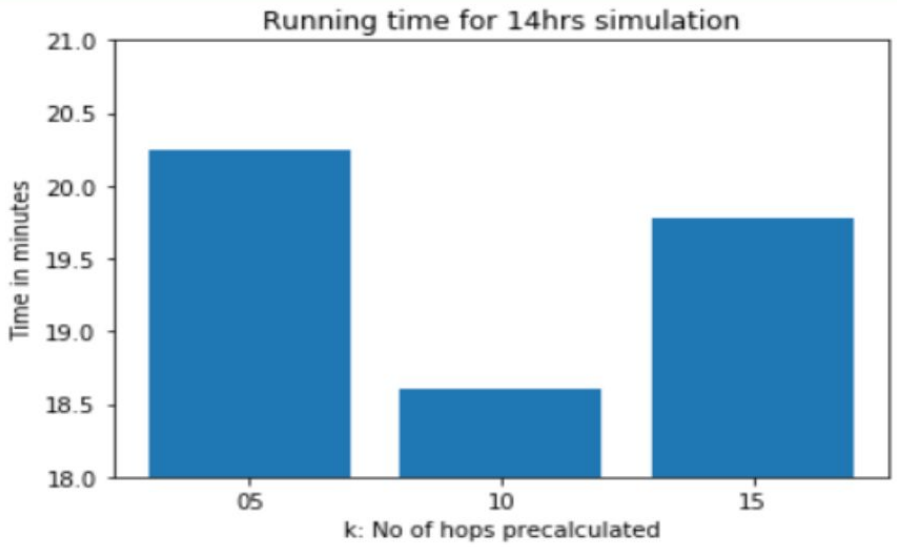
a) Counter-intuitively, the weighted model reduced the average % percentage to 68.3%.

b) The weighted model behaved well by reducing the average search and idle time giving extra weight to popular and zones giving trips with longer durations.

Tuning by Pre-calculated hops



We experimented our application with a various number of precalculated hops. This plot shows the comparison among them in terms of search & idle time. There was a negligible difference among the results since precalculating extra hops would affect navigation for edge cases, such as calculating the future path of 'k' hops at the end of a timeslot.



Navigation for K-hops runs in parallel for all the cabs

Tuning the number of precalculated hops (i.e. ‘k’) resulted in a significant difference in the running time. For a smaller ‘k’, the paths to be traversed need to be calculated more frequently, and vice-versa for a larger ‘k’. As can be observed from the graph, there is a drastic drop in the running time from 5 to 10 hops, since the pre-calculations are done half frequently. However, for k=15, the running time increases because the path is invalidated due to change in zone scores as per time slot.

Technologies / Third Party API Used

1. Preprocessing & ML
    - Python - flask, numpy, pandas, scikit-learn
2. Zones' details storage - JSON
    - NodeJS
    - Uber H3 JavaScript API (well documented)
3. Simulation System -
    - Java 8 - Parallel Streams
4. Deployment - AWS EC2
    - m5a.4xlarge - 64 GB RAM, 8 cores

References

[1] “The Rich and the Poor: A Markov Decision Process Approach to Optimizing Taxi Driver Revenue Efficiency” · Huigui Rong · Xun Zhou · Chang Yang · Zubair Shafiq · Alex Liu

https://www.biz.uiowa.edu/faculty/xzhou/files/cikm2016.pdf

[2] “Probabilistic spatio-temporal resource search” Qing Guo · Ouri Wolfson

https://www.cs.uic.edu/~wolfson/other_ps/geoinformatica2017.pdf

[3] ACM SIGSPATIAL Cup 2019: https://sigspatial2019.sigspatial.org/giscup2019/problem

[4] H3: A Hexagonal Hierarchical Geospatial Indexing System: https://github.com/uber/h3

[5] Amazon EC2 M5 Instances: https://aws.amazon.com/ec2/instance-types/m5/