On-Demand Transportation Scheduling

Final Report

Weini Yu
weiniyu

Zhouchangwan Yu
zyu21

Yutian Li
yutian

December 15, 2017

Contents

1 Introduction

As the mobile and telecom technologies advance, there has emerged many on-demand service platforms. One of them is on-demand transportation service platforms like Uber and Lyft that match riders (demand) with drivers (supply). This type of model enhances efficiency and adds job liquidity to the labor market and has a great positive impact on the economy as a whole.

One fundamental problem these platforms and services all need to solve is how to effectively match the customer requests with the vehicles, given the distances, travel times, request density within a range and many more factors. Our project aims to develop an intelligent model that provides a solution to a simplified version of this scheduling problem: the single vehicle pick-up and delivery problem (SVPDP)[1].

2 Related Works

Artificial intelligence has been widely applied to on-demand transportation problems. For example, a market formation algorithm was developed based on machine learning for Liftago, a mobile app that connects customers and taxi driver by taxi request, matchmaking, accept & bid, and driver selection[2]. With the proposed SImple Data-driven MArket Formation (SIDMAF) algorithm, the authors have improved the efficiency of matchmaking by increasing the average ratio of “accepts” per ride order, and decreasing the average number of selected drivers per ride order. Another blog post discussed the Simulated Annealing algorithm that adds randomness to the exploration and enhances their TransLoc OnDemand system[3]. Although the goal and the approach of these projects are different from that of our project, there are aspects that we could refer to, such as how they frame the on demand transport problem and extract key features.

3 Task Definition

As mentioned above, to not overcomplicate the system, we only look at the pick-up and delivery problem of a single vehicle and without considering carpool situation. On a given map, the initial position of the driver can be anywhere. The driver is given a snapshot of current rider requests nearby whenever it is time to pick up a new rider, which is at start time of the game or the end time of each trip (when the driver just dropped off a rider). From the list of requests the driver selects the next customer to pick up. The list of requests and locations are dynamic so at any time the snapshot may look different. At the end of each trip, the driver receives a reward.

The driver's total driving time is fixed, but the travel time and reward for each trip is different. The goal is to maximize the total reward of a driver within the driving period by balancing between trip fare, travel time, and rider satisfaction.

4 Data

4.1 Data Preprocessing

The above task requires information about locations, travel times between locations and trip fares. We are able to obtain all the data we need from Uber Movement and Developers API.

Locations (a.k.a. zone indices): The map of the city of Washington D.C. from Uber Movement is discretized into 558 zones (referred to as locations from now on). We pick the first n zones as our locations.

Travel times: We download the travel times between location pairs during the second and third quarter of year 2017. We pick the arithmetic mean travel time as the travel time between locations and keep the data in a $n \times n$ matrix. The distribution of travel times of the second quarter is shown in Figure 1.

Coordinates: In order to obtain the trip fare from Uber Developers API, we need the geolocation coordinates of the locations. We download the location-coordinates JSON data and calculate the arithmetic mean longitude and latitude within zones as the coordinates of the n locations.

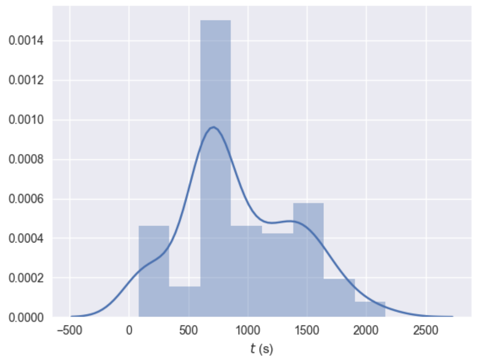

Figure 1: Distribution of traveling times.

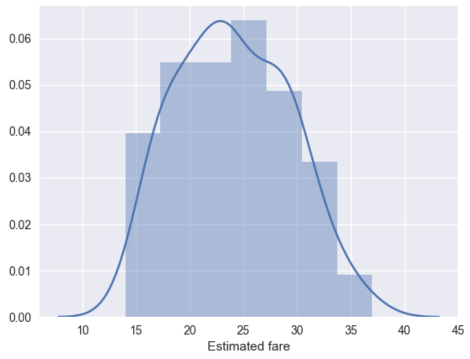

Figure 2: Distribution of estimated fares.

Fares: The fares between the locations are requested form Uber Developers API. Given the longitudes and latitudes of any two locations, the low estimate and high estimate of the fare are generated for various types of vehicles. Here we use the average of the fare for UberX and keep the data in a $n \times n$ matrix. The distribution of fare estimates are shown in Figure 2. We have obtained two complete sets of data, one of location size 10 ($n = 10$) from the second quarter of year 2017, one of location size 40 ($n = 40$) from third quarter of year 2017. Although we would like to increase the size of our map, we are limited by the max number of fare queries Uber Developers API allows during a period of time. We use these two sets of data to train our model and compare the results.

At the end of data preprocessing stage, we generate a `city.p` pickle file that has everything we need about the environment. It has a list of locations, travel times between locations, fare estimates, and coordinates of the locations.

4.2 Request Generation

A list of rider requests nearby is presented to the driver dynamically whenever it is time to pick up a new rider. We want to generate the requests in a way such that some locations are inherently better (e.g. more traffic flow). So the source location is randomly sampled with lower location indices having a higher weight. The destination location is sampled uniformly randomly. By generating requests in this nonuniform way, we create an imbalance between locations. We hope that the driver will figure out the underlying relationships between states, and use this knowledge to achieve higher rewards. The probability is visualized in Figure 3.

5 Baseline and Oracle

Based on the data we obtained and processed, we have implemented a baseline and an oracle.

Baseline: Driver randomly picks one request at each step.

Oracle: Driver knows the true travel times and reward for all the requests, and picks greedily according to it.

6 Model

We approach this task using a Markov decision process (MDP) model, with the intuition that at a certain location (state), the driver could choose to take one of the requests that come in (action). For each action the driver receives a reward. At the end of the day, the driver has a total reward for all the rides he or she

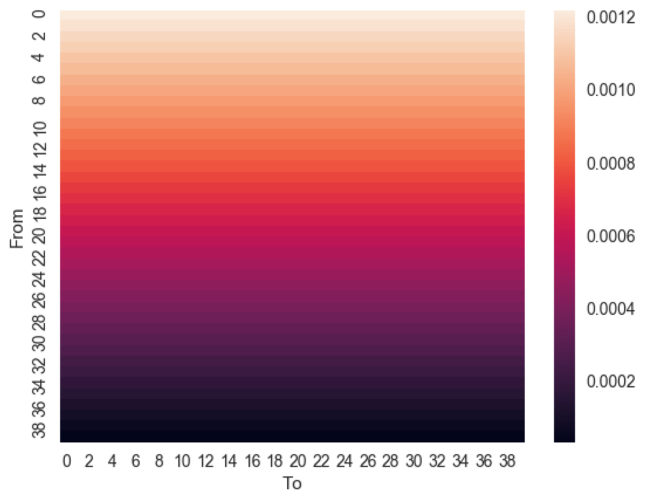

Figure 3: Probability of source and destination locations.

takes. Through solving this MDP, the driver can figure out which request to take at a location in order to maximize the total reward.

6.1 State

We originally had states that include the driver's current location and a list of requests at the location. However, this makes the state space very large (40^{11} for 40 locations and 5 requests) which is not desired. Also, the relative order of requests does not really matter.

Therefore, we simplify the state representation such that it only contains the driver's current location, a zone index number, reducing the state space to 40. For example, `state=1` means that the driver is at location 1. Note that we are not losing anything by not including requests. Requests are encoded into actions.

6.2 Action

Since state is the current location the driver is at, the action would be one possible request the driver receives, which is a tuple `(source, destination)`, where source is the pick-up location and destination is the drop-off location. In our specific model, the driver is given 5 requests to choose from at the start of the game and the end of each ride. So at any step, there are 5 legal actions. The driver must choose one from them.

6.3 Reward

The reward of each action is defined as (fare $-\ \alpha \cdot$ travel time). The intuition here is that gas cost needs to be subtracted and the longer the travel time is the less happy the customer is so there is the negative impact of travel time on the reward. The constant α balances fare and travel time, and can be seen as a measure of productivity. In our specific model, α is chosen to be $\frac{1}{150}$.

7 Algorithm

7.1 Q-learning

For our agent, we use Q-learning with function approximation to solve this problem. For a smaller sized problem we could run exhaustive search, but it is clearly not scalable. We choose Q-learning instead of TD learning, because intuitively states do not matter that much in our model. Though some states are inherently better due to the request generation probability, actions are much more important. Besides, if a driver takes a request and returns to the same location, the state is the same but clearly the total reward is not.

For function approximation, we manually select a few features and use linear regression to predict the Q value.

7.1.1 Features

We pick the following features.

Distance: The travel distance is one of the main concern for the drivers. Here we define two features related to the travel distance, `dist1`, the distance from current location to source, and `dist2`, the distance from source to destination.

Zone: Some locations are more attractive to people than others, so the drivers may have preference on where they would like to go. Each location as current location, source, or destination are all considered as features.

We define indicator features

$$\mathbf{1}\{\text{current location} = z\},$$

$$\mathbf{1}\{\text{source location} = z\},$$

$$\mathbf{1}\{\text{destination location} = z\}$$

for all z's.

7.1.2 Training

Using features described above, we learn the respective weights using Gradient Descent. We update weights using exponential weighted moving average as $w_i \leftarrow w_i - \eta(Q(s,a) - r)\phi_i(s,a)$, where $\eta = 0.1$. Note that we set discount $\gamma = 0$. This is because there is no way for the driver to know the legal actions (requests) available at future states. They are randomly generated only when the driver gets to a new state. $\max_a Q(s,a)$ is hard to define, hence $\gamma = 0$ for simplicity.

8 Example

For a concrete example, we assume the driver is at location s. Five requests (s_i, d_i) for $1 \leq i \leq 5$ are dispatched to the driver. The driver evaluates $Q(s,(s_i,d_i))$ for all five requests using linear estimator $\sum_i w_i\phi_i(s,a)$ and picks the one with the highest estimation. The driver takes this action. Then the reward r is given to the driver, who in turn uses $w_i \leftarrow w_i - \eta(Q(s,a) - r)\phi_i(s,a)$ to perform Gradient Descent to update the weights w.

9 Results

9.1 Model Performance

The model runs for a driving period of 12 hours. Figure 4 shows our results on the large dataset. Exponential moving average was performed to smooth the curve a little.

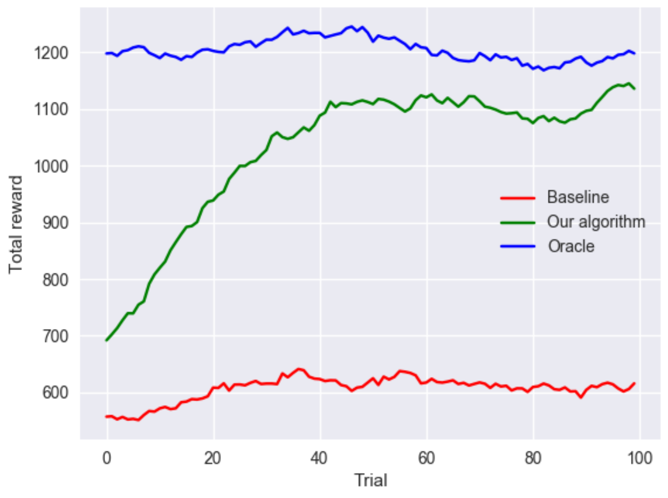

Figure 4: Rewards obtained by our algorithm, baseline, and oracle on large dataset ($n = 40$).

At first our algorithm acts randomly because all the weights are just initialized. It then ramps up quickly and learns the Q function. After 40 trials, it has already received almost twice the reward as that of the baseline. The rewards obtained through after convergence is shown in Table 1. Because of the probabilistic nature of the problem, baseline and oracle both fluctuate a lot. We can spot the slight raise around trial 90 for our algorithm. It does not represent a jump from the local optima but only the fluctuations of data. The overall trend is easy to spot. Eventually, our algorithm almost converges to a local optimal solution: achieving 90% higher rewards than the baseline, and only about 5.0% worse than the oracle.

Table 1: Total rewards on large dataset ($n = 40$).

Algorithm	Baseline	Our algorithm	Oracle
Trial 1	600	700	1200
Trial 20	600	950	1200
Trial 40	600	1140	1200
After convergence	600	1140	1200

Why can our algorithm achieve such good results? Q-learning is well suited for the task, as state does not matter much because the driver may return to the same spot with drastically different accumulated rewards. Instead, state-action pair conveys much more information. Also, relative distance as a feature captures the correlation of distance, fare, and travel time well.

Note that there is no separation of training and testing datasets. This is because for each trial, all the requests are generated on-the-fly. So there is no need to reserve a portion of the dataset for testing purposes. Also we want the driver to learn about a specific map, we do not consider the case of transferring knowledge to other cities.

9.2 Runtime Performance

Our algorithm runs under 5 seconds with negligible memory consumption. This is comparable to the baseline and oracle algorithms, both using only constant time per step. Because function approximation is used in the model, the actual information stored is greatly reduced. By handcrafting multiple features, the model tries to fit the underlying complex distribution with minimal loss. So after all, we are not storing or computing many more features than the baseline nor the oracle. In some sense this is similar to a dimensional reduction technique.

9.3 Error Analysis

We can further analyze the results by inspecting the weights of various learned features. We provide features `dist1` and `dist2` to describe the distance from the driver's current location to the rider's pick-up location and the distance from the pick-up location to the rider's drop-off location. In one specific run, our algorithm learned the weight of `dist1` to be -13.95 and the weight of `dist2` to be 324.30. The first weight means that the driver should minimize the distance to pick up, which is very intuitive. But this cost is greatly outweighed by the distance of the actual ride. Even though longer rides take more time, generally they are more profitable. The importance of `dist2` is higher than that of `dist1` because of the relative scale of values.

One of the underlying assumption is that some locations are inherently better than others by having more trips starting from there. So if the driver is at some superior location, it should be a shorter distance to pick up the next rider. We model this distinction using the feature describing the destination location of a trip named `dest_i`. Ideally, the driver would want to end up in a busier location because of the trip generation distribution. As can be seen from Figure 5, there is a vague trend that some locations are better; and the model has learned that locations with lower indices are better by giving higher weights. However, generally there is a lot of noise; the relationship is not exact.

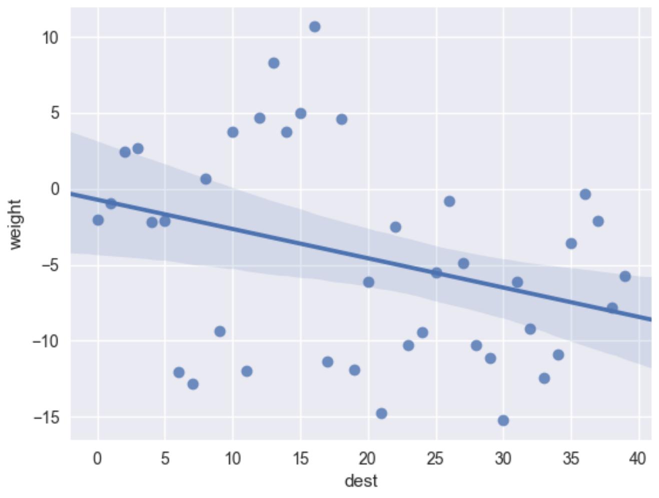

Figure 5: Feature weights for destination locations.

After our algorithm converges after around 40 trials, there is a constant gap between us and the oracle. An educated guess for why the gap exists is the bias in our selected features. We use l^2 Euclidean distance as a feature. But it cannot fully capture the real underlying travel time between spaces. There is definitely a strong correlation where places further apart takes longer to travel, while the property where crowded areas takes longer to traverse cannot be represented by this feature. Due to this inherent bias in our algorithm, the modeling error prevents us from achieving full oracle rewards.

9.4 Scalability

Besides basic runtime and model error analysis, we want to test the scalability of our algorithm.

We run the same experiments on the small dataset ($n = 10$), with both baseline and oracle. Runtime performance does not change that much. It completes under 3 seconds with negligible memory consumption, on par with baseline and oracle algorithms. The large dataset has 4 times number of states, and hence 16 times number of possible actions. Function approximation is able to catch the main relationship between states and actions in both datasets. So we do not see a 16 times memory usage or performance difference. As for total reward earned, we can see from Figure 6 that the algorithm converges after 40 trials as well. The result is very close to oracle; the rate of convergence is similar either. The overall shape and trend is like that of the larger dataset.

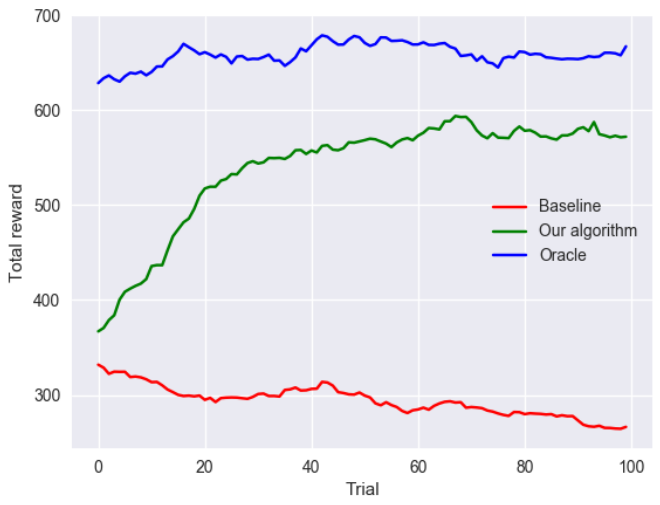

Figure 6: Rewards obtained by our algorithm, baseline, and oracle on small dataset ($n = 10$).

Therefore, we believe it is fair to say that our algorithm do scale very well with large number of states. While a more exhaustive algorithm might give us better results, they usually do not scale, resulting in at least quadratic time and space consumption.

9.5 Ablation Study

To train the model, we have used multiple handcrafted features. This leads to our second question: what features are not necessary?

To answer the question, we have conducted ablation study on the features using the small dataset($n = 10$). As mentioned above, the full set of features contains both locations and distances. Features involving locations describe the preferences for current, pick-up, and drop-off locations. Features involving distances mainly describe preferences for the distance to pick up and drop off the rider. So we test with location related features alone, and also with distance related features alone. The total rewards obtained after convergence is listed in Table 2. The results are shown in Figure 7.

Table 2: Total rewards after convergence for different feature sets.

Algorithm	Full	Only distance	Only location
Total rewards	582	562	373

With only location related features, we are able to reach almost the same level of rewards as that of full features. But with distance related features alone, the model is hardly learning any useful information and not able to improve.

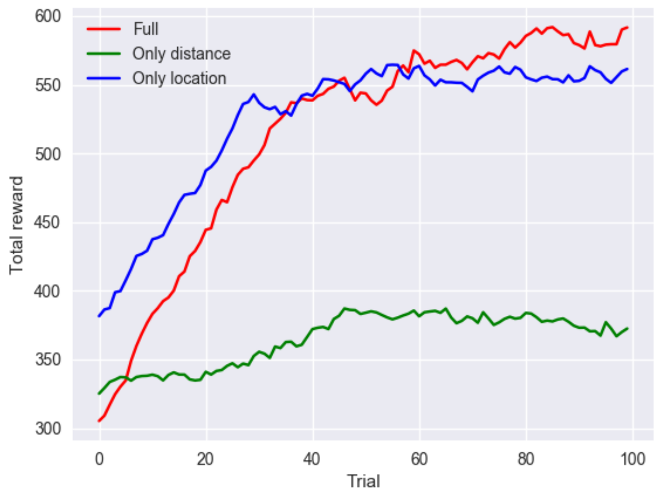

Figure 7: Rewards obtained by different feature sets.

9.6 Future Cost

As mentioned in our algorithm, we set $\gamma = 0$ for Q-learning. This poses the question whether future costs are factored into the model at all. But nothing is lost. To elaborate, the cost on the future in our model is the time taken to handle a single request, since the total time of driving is limited. So if we had set $\gamma \neq 0$, the model would have learned to favor quick requests a little more so more requests could be handled in total. But even though we set $\gamma = 0$, we added $(-\alpha \cdot \text{travel time})$ to the reward for each step. By adding this term, we add the cost of the time taken to finish this request, which is essentially the future cost. So in this sense even though we set $\gamma = 0$, we still factor into future costs. And also through results, we do observe that the total rewards is improved.

10 Discussion

10.1 Carpool

We want to extend our model to include the carpool situation, where the driver can have more than one rider in the car at the same time. Suppose we allow the driver to take at most 4 riders in the car at the same time. Rather than only presenting the driver a list of requests at drop-off, we interrupt the driver according to a Poisson process to give the option to pick up more riders on the way. Our map still has 40 locations.

State: Now the driver has to keep track of the destinations of all the riders currently in the car. To make sure a rider does not sit in the car forever, each rider's time in the car since pick-up also needs to be encoded into the state. So instead of just having the driver's current location, we need to extend our state to include an ordered list of `(destination, time_since_pick_up)` pairs of maximum size 4. The order is to indicate which rider to drop off next. The `time_since_pick_up` is discretized into buckets with domain size k. This increases the state space from 40 to $40^5 \times k^4$.

Action: In the case without carpooling, the driver selects one of the 5 rider requests. This action means to pick up *and* drop off a rider next. Here, at each decision point, the driver can choose to take another rider, which may result in a change of the next drop-off location, or not take another rider, which means the driver still tries to go to the drop-off location of the first request in the list.

Reward: Our previous reward function (fare $-\alpha \cdot$ travel time) is mostly from the driver's point of view. In the carpool case, however, it's important to make sure that all riders can get to their destinations in a reasonable amount of time. So we need to add an extra penalty on the time a rider spends in the car.

10.2 Nash Equilibrium

We also consider extending our model to solve multiple vehicles pick-up and delivery problem. We would like to find the optimal strategy to coordinate all drivers according to all riders' requests. Then there will be two situations, when demand is greater than supply, and when demand is less than supply.

When demand is greater than supply, drivers can take actions independently without competition. The global optimal strategy for maximizing the reward is each driver applying the individual optimal strategy, as we described above. However, this would lead to undesired situations for riders. For example, a rider far away from the city center may not be picked up by any driver for a long time, as drivers have many better request options from one popular place to another.

When demand is less than supply, drivers must compete against each other to take the requests. Then the optimal strategy for the drives are different from the previous situation. For example, drivers may prefer to stay around the city center as more requests are likely to be generated from there. On the other hand, as there is more supply, the riders sending out the requests will be quickly picked up by some driver, while other drivers will be idle. As a result, some of them may tend to move to other areas with fewer requests but less competition as well. Eventually, it will lead to an equilibrium where the request density and driver density are balanced.

11 Conclusion

We present the problem of on-demand transportation scheduling and offer a Q-learning algorithm to solve it. In addition to the algorithm, we also provide a general framework for investigating this problem and testing out different algorithms. Our algorithm shows great performance, both in total rewards earned and in runtime. It almost achieves the result of the oracle, and runs in around the same amount of time with negligible memory assumption.

We demonstrate the scalability of the algorithm. We also point out that the most important features are those related to locations. We further discuss the possibilities of carpooling and various Nash equilibriums in this problem. We hope this will spur a wider range of research on the topic.

Our code is uploaded to GitHub at https://github.com/hotpxl/uber-agent.

There is also a runnable version on CodaLab at
https://worksheets.codalab.org/worksheets/0x9463b2a06f2545d6830e3a67a3fd8fd9/.

List of Figures

List of Tables

References

[1] Manar I. Hosny, Christine L. Mumford. *The single vehicle pick-up and delivery problem with time windows: intelligent operators for heuristic and metaheuristic algorithms*. Journal of Heuristics, 2008.

[2] Jan Mrkos, Jan Drchal, Malcolm Egan, Michal Jakob. *Liftago On-Demand Transport Dataset and Market Formation Algorithm Based on Machine Learning*. 2016.
`https://arxiv.org/abs/1608.02858v1`

[3] Michael Fogleman. *Ridesharing Algorithms in TransLoc OnDemand*.
`https://techlog.transloc.com/ridesharing-algorithms-in-transloc-ondemand-7acfeddbfe1f`