Calyber: A Shared Rides Pricing and Matching Game

IEOR C253 Spring 2025*

March 15, 2025

1 Background and Introduction

Ride-sharing platform Calyber offers shared rides services to riders in Chicago. This is a carpooling service that matches two riders that have similar routes together to share a ride, thereby reducing the driving cost as well as offering lower prices to riders. Promoting shared rides is crucial for a sustainable future of urban transportation, reducing traffic congestion and carbon emission.

As we discussed (or will discuss depending on when you read this) in class, a key challenge in operating ridesharing platforms is to price and match incoming riders. For shared rides, these are more complicated as riders must be matched with each other in addition to being assigned a driver. While we primarily focused on the matching challenge in class, this game emphasizes pricing, assuming there are always enough drivers available where needed.

Riders arrive randomly over time. The platform sets a price for each arriving rider, and rider can choose to accept or reject the price. If accepted, the platform will try to match the rider with another rider, allowing them to share a ride to their destinations. The platform earns revenue from the total prices paid by them while covering the driving cost of picking them up and dropping them off. If no suitable match is available, the platform can defer the matching. However, riders have a limited waiting patience, and if exceeded, the platform misses the matching opportunity and must dispatch the rider solo, leading to higher costs.

In this game, you will play the role of the platform. The goal is to maximize profit by setting the price for each incoming rider as well as matching riders who accepted the prices (we call them requests or riders converted). The total profit is calculated as the total revenue collected from all riders minus the total driving costs of serving them. But here's the challenge—your decisions come with tricky trade-offs: for pricing, you have to properly assess the potential cost of serving the rider as well as balancing rider's willingness to pay. Assessing the cost is nontrivial because it depends on the matching outcome which might be uncertain at the time of rider arrival. For matching, the key

*Prepared by Yifan Shen from UW Seattle and Chiwei Yan from UC Berkeley. Contributions to the teaching materials were also made by Manuel Martinez Garcia, Jennifer Yu-Chen Huang, Santiago Karam Padilla and Kenny Wongchamcharoen. The authors thank Julia Yan, Shuo Sun and Venkatesh Ravi for suggestions.

trade-off involves deciding whether to match immediately or wait for future riders who may align better with the route. However, extended waiting times risk exceeding rider' patience thresholds, resulting in low-efficiency solo rides.

1.1 Definitions of Pricing and Matching Policies

We first define the pricing and matching policies. These policies are mappings from the system **state s** to pricing and matching decisions respectively. The state $\mathbf{s}$ is defined as **the set of waiting requests** (that is, the riders who have already accepted to pay the prices but are not matched and dispatched yet). The pricing and matching policies are thus defined as:

- **Pricing policy:** Given a state $\mathbf{s}$ and a rider i, the pricing policy $\psi_i(\mathbf{s})$ determines the price $p_i \in [0, 1]$ offered to rider i arriving at state $\mathbf{s}$.
- **Matching policy:** Given a state $\mathbf{s}$ and a rider i that converts at state $\mathbf{s}$, the matching policy $\phi_i(\mathbf{s})$ decides which waiting request $j \in \mathbf{s}$ to be matched with request i, or does not match and lets rider i wait first.

1.2 Game Dynamics

We proceed to describe the dynamics of the game.

Dynamics from the rider's perspective. The dynamics from the rider's perspective is described below and depicted in Figure 1. Imagine that a rider i arrives on the platform entering her origin O_i and destination D_i. The platform then quotes an upfront price $p_i \in [0, 1]$ (\$/mile). Based on the quoted price p_i and the rider's own willingness to pay $v_i \in [0, 1]$ (\$/mile) (the value of v_i is private and drawn from some distribution), this rider decides whether to accept the price and make a request (i.e., convert) or to leave the platform directly (i.e., not convert). Specifically, if $v_i \geq p_i$, then she converts; otherwise, she does not. Assume that ℓ_i is the trip length (in miles) if rider i rides solo. Then the amount she pays to the platform is $p_i \ell_i$.

For rider i just converted, the platform makes the matching decision. If the platform decides to match rider i with another waiting request, then the two riders are sent on a shared ride immediately. The platform can also choose to let rider i wait in the system first. She will stay in the system for a limited amount of time following an *exponential* distribution with a rate $\theta_i > 0$. If she is not matched with another rider before her patience explodes, the platform will have to dispatch her solo in an individual ride. Every time a ride is dispatched, the platform needs to pay a driving cost at a rate of c (\$/mile). Assuming ℓ_{ij} is the total length of a shared ride involving rider i and j (and let $\ell_{ii} = \ell_i$), then the driving cost is $c\ell_{ij}$ if it is a shared ride involving rider i and j, and the cost is $c\ell_i$ if it is an individual ride for rider i. Please see Section 4.3 for how ℓ_{ij} is calculated.

Dynamics from the platform's perspective. We also describe the dynamics from the platform's perspective and illustrate it in Figure 2. Assume that the system is in state $\mathbf{s}$ at some time t. Then there are two possibilities for the next event: either a new rider i arriving, or a waiting request $i' \in \mathbf{s}$ reneges. We discuss these two cases.

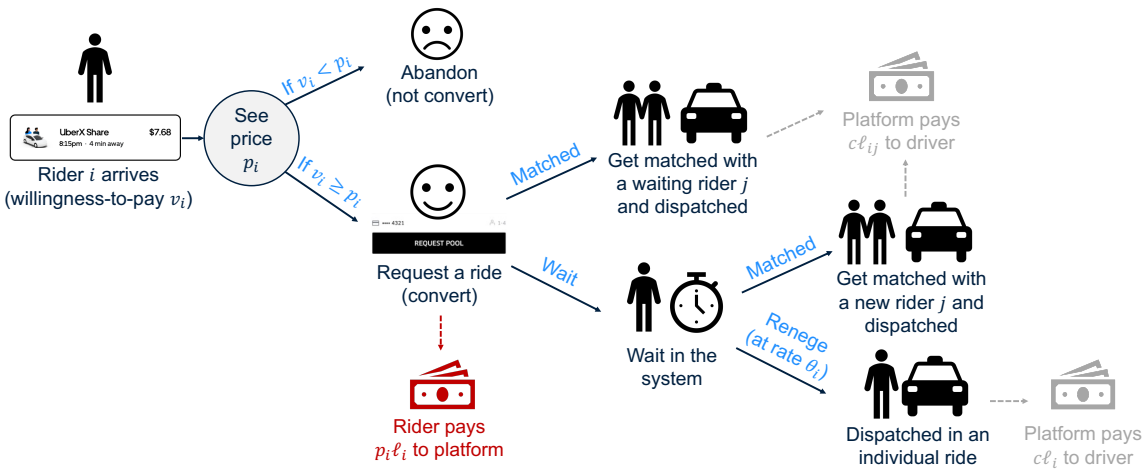


Figure 1: Dynamics from the rider's perspective.

1. When the next event is rider i arriving, then the platform needs to determine the price p_i according to the pricing policy $\psi_i(\mathbf{s})$.
   - If rider i does not convert, then she leaves immediately. Nothing happens and state $\boldsymbol{s}$ does not change.
   - If rider i converts, then the platform collects a revenue of $p_i \ell_i$ and determines its match according to the matching policy $\phi_i(\mathbf{s})$. If the platform decides to match rider i with an waiting request $j \in \mathbf{s}$, then the platform pays a dispatching cost of $c\ell_{ij}$, and the state is updated to be $\mathbf{s} \backslash \{j\}$. If instead the platform decides not to immediately match rider i with any currently waiting requests, then she waits in the system and the state is updated to be $\mathbf{s} \cup \{i\}$.
2. When the next event is a request $i' \in \mathbf{s}$ running out of her patience, then she is dispatched solo and the platform pays a dispatching cost of $c\ell_i$. The state is updated to $\mathbf{s} \backslash \{i\}$.

2 Project Description

This section is a detailed overview of the project: what you are given, what you need to do, and how your policies will be evaluated. All relevant files are in the GitHub repository, which you can access by accepting the assignment at the Github Classroom link `https://classroom.github.com/a/e2kqT6e0`. Once you create or join a team, you'll see your team's repository, where you'll also submit your work.

2.1 What you are given

You are given a set of historical rider arrival and request data (under the default pricing and matching policies described in Section 4.2) as the training data set and the infor-

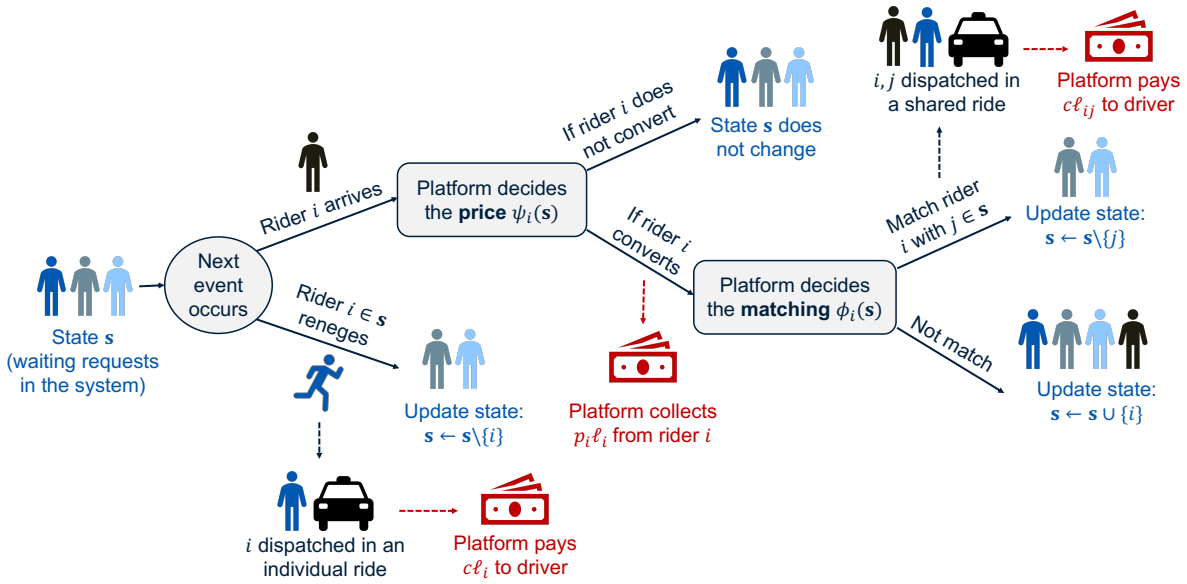


Figure 2: Dynamics from the platform's perspective.

mation of Chicago community areas. The driving cost rate is set to be $c = 0.7$ ($/mile) throughout the game.

2.1.1 Training data

The training data set contains the information of 11,788 riders who arrived at the platform in the past. The time span is an one-hour time window across six weeks. Each rider has the information of the arrival time and pickup and dropoff locations. The default pricing and matching policies were implemented for these riders, so the training data also contains the prices quoted to each rider, whether they converted or not, their waiting time, and their matching outcomes.

The data is stored in `\data\training_data.csv`. The first row is the header, and each row after represents an arriving rider. Each rider has the following fields:

- `rider_id`: Rider's unique ID (integer, 0–11787).
- `arrival_week`: Week of rider arrival (integer, 1–6).
- `arrival_time`: Arrival time within the hour (in seconds, 0–3600).
- `pickup_area`, `dropoff_area`: Index of the rider's pickup and dropoff area (1–76).
- `pickup_lat`, `pickup_lon`: The exact latitude and longitude of the pickup location.
- `dropoff_lat`, `dropoff_lon`: The exact latitude and longitude of the dropoff location.

- `solo_length`: Solo ride distance (in miles), which is the haversine distance between the pickup and dropoff locations of the rider.
- `quoted_price`: Quoted price per mile ($/mile) under the example policy (float, 0–1).
- `convert_or_not`: Whether the rider converts (1 for yes, 0 for no) under the example policy.
- `waiting_time`: Rider's waiting time (in seconds), the time difference between the arrival time and the dispatching time of the rider. `nan` if the rider does not convert.
- `matching_outcome`: Matched rider's ID if matched, otherwise `nan`.

2.1.2 Area information

The Chicago city is divided into 76 community areas. Each rider is picked up and dropped off at the centroids the community areas. You are given the latitude and longitude of the centroid of each area. The data is in `\data\area_lat_lon.csv` file. Each row represents an area. The columns are:

- `area`: Community area index (from 1 to 76).
- `lat`, `lon`: Latitude and longitude of the area's centroid.

2.2 What you need to do

You need to write your own pricing and matching policies in `student_policies.ipynb`. Please follow the instructions:

1. Before you start writing the code, your team should have a unique name. Let's suppose your team name is "Awesome Team".
2. Change the `TEAM_NAME` constant in your .py file to your team name in camel case, e.g., `AwesomeTeam`.
3. Write the code for your pricing and matching policy in the `pricing_function` and `matching_function` method of `StudentPricingPolicy` and `StudentMatchingPolicy` classes.
4. Export `student_policies.ipynb` into a python file `TEAM_NAME_Policies.py` using the function provided in the end of the notebook.

Several additional important notes.

- **Time limit:** there are strict time limits on how long your policies are allowed to return a decision. **The execution time limit is 0.05 seconds for every call of the pricing or matching policy**. If the running time exceeds the limit, then the algorithm is terminated, and the pricing decision would be $p = 1$, or the matching decision would be not matching with anyone. Hence, you should likely not include your training logic in these functions — train your policy offline and paste the training results in these two functions.

- **Package:** you are not allowed to use additional python packages (e.g., Gurobi) other than those already loaded in `student_policies.ipynb`. However, as mentioned in the previous bullet point, you can train your policy offline using any packages you want and paste the results into these functions.
- **Code format:** you are not allowed to change the input of the `pricing_function` and `matching_function`. The output should also follow the rules described in the template file.
- **Helper functions:** a helper function `populate_shared_ride_lengths()` to calculate ℓ_{ij} of a shared ride involving riders i and j is provided in `utils.py`, and an example is provided in `helper_functions_demonstration.ipynb` to help you understand how to use it. You can directly call it in your code. See Section 4.3 for a detailed description.
- **Policy test:** a testing function `test_policies()` is provided in `utils.py`. It helps you verify your policies satisfy the formatting requirement, so we anticipate no bugs when running your code.

2.3 How your policies will be evaluated

Your policies will be evaluated on a test data set. The test data set contains three different weeks of data collected from the same time window as the training data. The simulation logic of how your policies will be evaluated on the test set is shown in the pseudocode in Section 4.4, which is very similar to the dynamics described in Section 1.2 and Figure 2.

Your policies will be ranked according to the total profit collected in the test set. If several policies produce similar profit values, we would consider the following additional metrics to produce the final ranking.

- `throughput`: number of total requests (converted riders);
- `match_rate`: `matched_riders` / `throughput`;
- `conversion_rate`: `throughput` / total number of rider arrivals;
- `cost_efficiency`: 1 - `cost` / (`throughput` $* c$);
- `avg_quoted_price`: the average price ($ per rider, per mile) over all rider arrivals;
- `avg_payment`: the average price ($ per request, per mile) over requests;
- `avg_waiting_time`: the average waiting time (seconds) to get a dispatch over requests.

3 Project Logistics and Timeline

3.1 Logistics

You can work on the project in a team of *at most* three people, the same team you worked with for the Sport Obermeyer case. Please pick a team name (make sure it's different from other teams). Your `TEAM_NAME_Policies.py` (exported from `student_policies.ipynb`) is the only file you need to submit for evaluation. You can only submit one python file for each team. You will be ranked based on the performance of the test data set. Top teams will earn bonus scores and be invited to present how you develop your policies in the final lecture of the semester. Each team needs to submit a project report of 10 pages detailing your strategy.

3.2 Validation

To help you test and improve the performance of your policies, we provide a validation run before the final test. Your code will be run on a validation data set and the simulation results will be returned to you, which is a csv file of the data of each rider (the same format as the training data) plus the timeout information as well as a csv file reporting different performance metrics. A ranking will also be produced across teams based on the validation performance. You can use the validation results to debug your code and improve its performance. The validation run is optional, though highly encouraged, and its performance will *not* affect your final score.

3.3 Timeline

- Week 7: Calyber game released.
- Week 12: (Optional) Submit your code for validation to your team's repository.
- Week 13: Submit your code for the final evaluation to your team's repository.
- Week 14: Ranking announced.
- Week 14: Top teams present their projects in the class.
- Final Week: Submit the project report.

3.4 Staff

If you have any questions regarding the codebase, contact Jennifer Yu-Chen Huang (jenhuang@berkeley.edu), Manuel Martinez Garcia (manpazito@berkeley.edu), the GSI, or the professor for questions regarding developing your matching and pricing policies.

4 Appendix

4.1 Notations

Technical notation used in this document are listed here.

- O_i: the origin of rider i.
- D_i: the destination of rider i.
- v_i: the willingness-to-pay (\$/mile) of rider i. $v_i \in [0, 1]$.
- θ_i: the reneging rate of rider i.
- ℓ_i: the trip length (in miles) of the individual ride of rider i, i.e., the haversine distance between O_i and D_i.
- ℓ_{ij}: the shortest total length of the shared ride between rider i and j. $\ell_{ii} = \ell_i$.
- $\tilde{\ell}_{ij}$: the length of the shared part in the shared ride between rider i and j.
- c: the dispatching cost of rides (\$/mile).
- p_i: the price (\$/mile) quoted to rider i. $p_i \in [0, 1]$.
- $\mathbf{s}$: the state of the system, which is the set of waiting requests.
- $\psi_i(\mathbf{s})$: pricing policy.
- $\phi_i(\mathbf{s})$: matching policy.

4.2 Example Policies

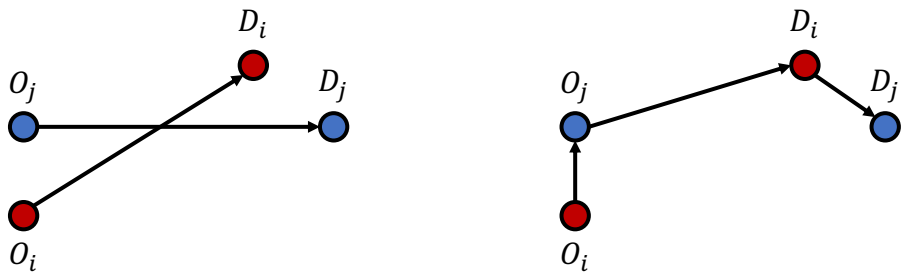
You are given one pricing and one matching policies as examples. The example pricing policy is an area-dependent static pricing policy such that the riders originating from the same area will have similar prices, and the areas with denser demands tend to quote lower prices. The example matching policy is a greedy policy that matches the incoming request to a waiting request with maximum shared length. The code is stored in the file `example_policies.py`, and the Jupyter notebook `example_pricing_policy_calculation.ipynb` shows how the example pricing policy is constructed.

- **Example pricing policy**:
  Let N_k be the number of riders originating from area $k, k = 1, ..., 76$, counted within the training set. This example policy lets the price for rider i originating from area $n(i)$ be $p_i = \frac{c}{2} + (1 - \frac{c}{2})(1 - \frac{\log N_{n(i)}}{\log(\max_k N_k)}) + \epsilon$, where $\epsilon \sim U(-0.1, 0.1)$ is a random noise. p_i is truncated to be within $[c/2, 1]$, so the lowest possible price is $c/2$ and the highest is 1.

- **Example matching policy**:
  If state $\mathbf{s} \neq \emptyset$, the arriving rider i is immediately matched with the waiting request $j^* \in \mathbf{s}$ which shares the longest nonzero length with i. Assume $\tilde{\ell}_{ij}$ is the length of the shared part in the shared ride between rider i and j, then $j^* = \text{argmax}_{j \in \mathbf{s}, \tilde{\ell}_{ij} > 0} \tilde{\ell}_{ij}$. Otherwise (either if there is no waiting requests, i.e., $\mathbf{s} = \emptyset$, or if no waiting requests share any length with the incoming rider, i.e., $\max_{j \in \mathbf{s}} \tilde{\ell}_{ij} = 0$), let rider i wait in the system first. The calculation of shared length $\tilde{\ell}_{ij}$ can be found in Section 4.3 below.

4.3 A Helper Function

The helper function `populate_shared_ride_lengths(origin_i, destination_i, origin_j, destination_j)` stored in `utils.py` calculates the driving cost and cost allocation of the shared ride involving any two riders i and j. For example, there are two riders i, j as illustrated in Figure 3a, where the origin-destination pair of rider i is (O_i, D_i) and that of rider j is (O_j, D_j). If they are matched together, the shortest distance is $O_i - O_j - D_i - D_j$ as in Figure 3b. The total length of this sequence is ℓ_{ij}.



(a) Not matched (b) Matched

Figure 3: Example of the shortest path and cost allocation of a shared ride.

The input of the function includes four arguments:

- `origin_i`: a tuple `(lat, lon)` indicating the latitude and longitude of O_i.
- `destination_i`: a tuple `(lat, lon)` indicating the latitude and longitude of D_i.
- `origin_j`: a tuple `(lat, lon)` indicating the latitude and longitude of O_j.
- `destination_j`: a tuple `(lat, lon)` indicating the latitude and longitude of D_j.

The output includes five terms:

- `trip_length`: the shortest length of the ride involving rider i and j (in miles). For example, in Figure 3b, it is the length of $O_i - O_j - D_i - D_j$. This is denoted by ℓ_{ij}.
- `shared_length`: the length that is shared by the two riders. For example, in Figure 3b, it is the length of $O_j - D_i$. This is denoted by $\tilde{\ell}_{ij}$.

- `i_solo_length`, `j_solo_length`: the length that is only taken by rider i (or j). For example, in Figure 3b, `i_solo_length` is the length of $O_i - O_j$, and `j_solo_length` is the length of $D_i - D_j$.

- `trip_order`: an integer from 0 to 4 denoting the the pickup and dropoff order of the shared ride in the shortest path. Define five constants `IIJJ=0`, `IJIJ=1`, `IJJI=2`, `JIJI=3`, `JIIJ=4`. For example, in Figure 3b, the optimal pickup-dropoff order is to pick up i - pick up j - drop off i - drop off j, so the `trip_order=IJIJ=1`.

4.4 Simulation Logic

The simulation logic of how your policies are evaluated on a data set is shown below.

Algorithm 1 Simulation

Require: Pricing policy $\psi_i(\mathbf{s})$, matching policy $\phi_i(\mathbf{s})$, driving cost rate c, set of arriving riders (where each rider i has features of origin O_i, destination D_i, williness-to-pay v_i, and patience)

`profit` $\leftarrow 0$ ▷ Initial profit.
for each week in the data set **do**
  $t \leftarrow 0$ ▷ Accumulated time during a time window.
  $\mathbf{s} \leftarrow \emptyset$ ▷ Initial system state.
  repeat
    Receive the next event: a rider i arrives, or arequest $i \in \boldsymbol{s}$ reneges
    $t \leftarrow$ The time when the next event happens
    if next event is rider i arriving **then**
      Quote price $p_i = \psi_i(\mathbf{s})$ ▷ Use *pricing policy* to decide the quoted price.
      if $v_i < p_i$ **then** rider i does not convert
      else
        Rider i converts
        `profit` $\leftarrow$ `profit` $+ p_i \ell_i$ ▷ Collect the revenue from the request.
        $j \leftarrow \phi_i(\mathbf{s})$ ▷ Use *matching policy* to decide the request to match.
        if j is `None` **then** ▷ No match.
          $\mathbf{s} \leftarrow \mathbf{s} \cup \{i\}$
        else ▷ Match i with j.
          $\mathbf{s} \leftarrow \mathbf{s} \backslash \{j\}$
          `profit` $\leftarrow$ `profit` $- c\ell_{ij}$ ▷ Deduct the dispatching cost of the shared ride.
        end if
      end if
    else if next event is a request $i \in \mathbf{s}$ reneging **then**
      $\mathbf{s} \leftarrow \mathbf{s} \backslash \{i\}$
      `profit` $\leftarrow$ `profit` $- c\ell_i$ ▷ Deduct the dispatching cost of the individual ride.
    end if
  until $t > 3600$ ▷ The one-hour time window closes.
  if $\mathbf{s} \neq \emptyset$ **then**
    `profit` $\leftarrow$ `profit` $- c\ell_i$ ▷ Remaining requests are all dispatched solo.
  end if
end for