Ridesharing Simulation to Explore Matching Algorithms

Kazuyuki Shudo
Kyoto University

Tsuyoshi Hasegawa
Kyoto University

Yuito Ueda
Kyoto University

Hiroshige Umino
newmo, Inc.

Masahiro Sano
newmo, Inc.

Keisuke Sogawa
newmo, Inc.

***Abstract*—In a ridesharing service, driver-rider matching affects the business in several ways. Not only does it cost money to find a route on a map, but it also has an impact on the business in terms of driver income and rider waiting times, and therefore their satisfaction. It is not practical to field test the matching. For example, an A/B test would take days. A simulator greatly accelerates matching experiments. We modeled ridesharing and developed a simulator based on it. The event-driven simulator enabled us to test matching algorithms with a variety of parameters while utilizing routing systems, such as OSRM. Tested matching techniques are back-to-back and reassignment, that determine the matching targets. The simulations showed that back-to-back reduced the average waiting time for riders by up to around 40% and reassignment reduced it by up to around 15%. It is also the contribution of this paper to show what ridesharing operators are thinking about in order to technically optimize their business.**

***Index Terms*—ridesharing, simulation, driver-rider matching**

I. Introduction

Ridesharing is a service that matches drivers and riders via smartphone applications. The service is available around the world, and in 2024 we launched our services in Japan. In ridesharing, matching drivers and riders is critical [1], [2], [3], [4]. Efficient matching that reduces rider waiting times leads to increased rider satisfaction, and if drivers can serve more riders per unit time, their income is likely to increase. For operators, matching algorithms with low computational cost, e.g. for route search, help to reduce financial expenses.

It is not realistic to conduct experiments on matching algorithms in the real service. Field testing takes significant time and financial costs. For example, an A/B test takes several days. In addition, field testing may impose undue disadvantages on both drivers and riders. A simulator greatly accelerates such experiments. We modeled ridesharing and developed a simulator based on it. The simulator enabled driver-rider matching experiments quickly and cost-effectively without affecting our real services. Specifically, it allows us to conduct experiments on matching algorithms with various parameters, such as numbers, positions and their changes of drivers and riders. Furthermore, by utilizing OSRM (Open Source Routing Machine) [5], an open-source routing system, the simulator achieves a realistic simulation.

The target techniques for simulation are *back-to-back* and *reassignment*. Back-to-back (B2B) includes drivers who have

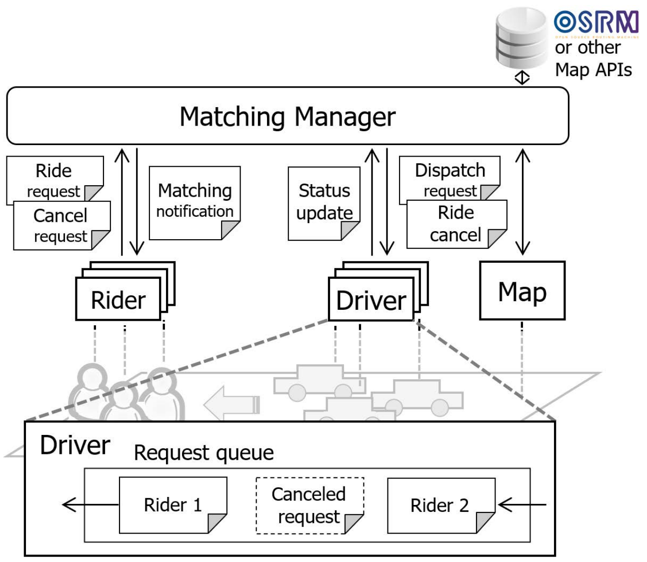


Fig. 1. Our model of rideshairng.

already been matched a rider as matching candidates. Reassignment allows reassigning a rider who have already matched to another driver. Simulations show that the B2B reduces the average waiting time for riders by around 40%. From another perspective, the same waiting time as with 35 drivers can be achieved with only 20 drivers using the two algorithms.

The contribution of this paper includes the followings.

- A design of ridesharing simulator, that is capable of experiments satisfying ridesharing operators' needs.
- Techniques for driver-rider matching, back-to-back and reassignment, that operators utilizes to technically optimize their business.
- How much the two techniques affect the interests of drivers, riders and an operator. E.g. around 40% reduction of waiting time for riders.

The following Section II and Section III describe our model of ridesharing and simulator implemented along the model. Section IV and Section V present the matching algorithms and their evaluation. Section VI shows related work and Section VII concludes the paper.

Algorithm 1 Map

```
Require: travel time calculation function: $f(x_1, y_1, x_2, y_2)$
1: procedure MAP(f)                                          ▷ Constructor
2:     $self.f \leftarrow f$
3: end procedure
4:
5: function GET_TRAVEL_TIME($x_1, y_1, x_2, y_2$)
6:     $time \leftarrow self.f(x_1, y_1, x_2, y_2)$
7:     return $time$
8: end function
```

II. A model of ridesharing

Our model of ridesharing to be implemented in the simulator consists of *Rider*, *Driver* and *Map* as shown in Figure 1. The simulator has a *Matching Manager*, that performs driver-rider matching.

Riders and Drivers appear on the Map by being *spawn*ed and disappear by being *despawn*ed along the given simulation scenario. In the real world, a spawn and a despawn correspond to the launch and the termination of the ridesharing application. A Rider issues a *ride request* with its current location and destination to the Matching Manager just after it is spawned. A Driver sends a *status update* to the Matching Manager periodically after it becomes available. The Matching Manager manages Riders and Drivers, and performs driver-rider matching at regular intervals. The Matching Manager sends a *matching notification* to the Rider and a *dispatch request* to the Driver when it *dispatch*es the Rider to a Driver.

A. Map

Algorithm 1 defines the Map in our model. The Map represents driver movement based on a external travel time calculation function f. The function takes two points on the map as arguments and returns the travel time. A simple example of the function is the straight distance between the two points divided by speed. The OSRM [5] based function is available in the simulator though other routing systems, such as Google Maps, can be supported. The function returns the travel time along routes calculated by the OSRM.

B. Rider

Algorithm 2 and a state transition diagram shown in Figure 2 define the Rider. A Rider is initialized with four parameters, spawn time t_1, despawn time t_2, spawn coordinate (x_1, y_1) and destination coordinate (x_2, y_2). The initial state is `UNMATCHED`.

When a Rider is spawned (line 8), it sends a *ride request* message to the Matching Manager. Upon receiving a *matching notification* (line 16), the Rider waits for the driver until it arrives, and then rides to the destination. At despawn time (line 12), the Rider sends a *cancel request* to the Matching Manager if it has not been dispatched to a Driver. This means that the Rider has given up boarding.

Algorithm 2 Rider

```
Require: spawn time: $t_1$, despawn time: $t_2$, spawn coordinate:
    $(x_1, y_1)$, destination coordinate: $(x_2, y_2)$
1: procedure RIDER($t_1, t_2, x_1, y_1, x_2, y_2$)              ▷ Constructor
2:     $self.spawn\_time \leftarrow t_1$
3:     $self.despawn\_time \leftarrow t_2$
4:     $(self.spawn\_x, self.spawn\_y) = (x_1, y_1)$
5:     $(self.destination\_x, self.destination\_y) = (x_2, y_2)$
6: end procedure
7:
8: when $self.spawn\_time$
9:     Send ride_request $(self.spawn\_x, self.spawn\_y, self.destination\_x, self.destination\_y)$ to Matching Manager
10: end when
11:
12: when $self.despawn\_time$
13:     Send cancel_request to Matching Manager
14: end when
15:
16: when receiving matching notification $d$
17:     Wait until Driver $d$ arrives
18:     Ride to coordinate $(self.destination\_x, self.destination\_y)$
19: end when
```

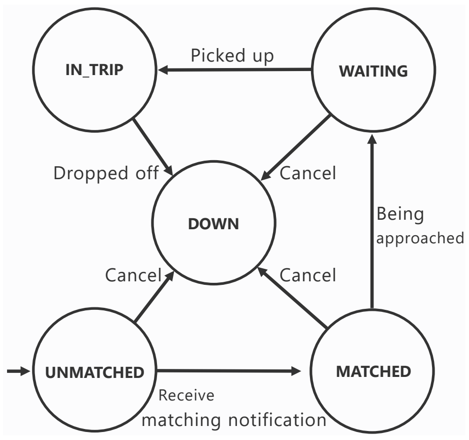


Fig. 2. State transition diagram of Rider.

C. Driver

Algorithm 3 and a state transition diagram shown in Figure 3 define the Driver. A Driver is initialized with three parameters, spawn time t_1, despawn time t_2 and spawn coordinate (x_1, y_1). The initial state is `DOWN`.

When a Driver is spawned (line 8) along a simulation scenario, it sends a *status update up* message to the Matching Manager. When it receives a *dispatch request* (line 16), it moves to the rider's location (line 25) and picks up the rider (line 26). Then, upon arriving at the destination (line 28), it drops off the rider (line 29). A Driver notifies the Matching Manager every time its state changes by sending *status update*

Algorithm 3 Driver

```
Require: spawn time: $t_1$, despawn time: $t_2$, spawn coordinate: $(x_1, y_1)$
 1: procedure DRIVER($t_1, t_2, x_1, y_1$)                    ▷ Constructor
 2:     $self.spawn\_time \leftarrow t_1$
 3:     $self.despawn\_time \leftarrow t_2$
 4:     $(self.x, self.y) = (x_1, y_1)$
 5:     $self.request\_queue \leftarrow []$
 6: end procedure
 7:
 8: when $self.spawn\_time$
 9:     Send status_update_UP $(self.x, self.y)$ to Matching Manager
10: end when
11:
12: when $self.despawn\_time$
13:     Send status_update_DOWN to Matching Manager
14: end when
15:
16: when receiving dispatch request $r$
17:     if $self.request\_queue$ is empty then
18:         $self.drive(r)$
19:     else
20:         Add $r$ to $self.request\_queue$
21:     end if
22: end when
23:
24: procedure DRIVE($r$)
25:     Send status_update_PICKUP to Matching Manager
26:     Drive to coordinate $(r.spawn\_x, r.spawn\_y)$
27:     Pick up Rider $r$
28:     Send status_update_IN_TRIP $(r.spawn\_x, r.spawn\_y)$ to Matching Manager
29:     Drive to coordinate $(r.destination\_x, r.destination\_y)$
30:     Drop off Rider $r$
31:     if $self.request\_queue$ is empty then
32:         Send status_update_UP $(self.x, self.y)$ to Matching Manager
33:     else
34:         $self.drive(self.request\_queue.dequeue())$
35:     end if
36: end procedure
37:
38: when receiving ride cancel $r$
39:     Remove the Rider $r$ from $self.request\_queue$
40: end when
41:
42: loop
43:     Send status_update_coordinate $(self.x, self.y)$ to Matching Manager
44:     Sleep for a short time
45: end loop
```

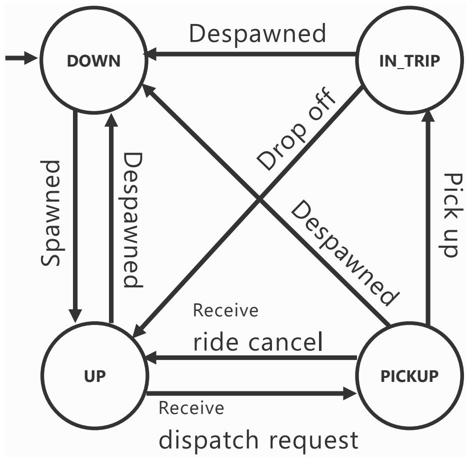


Fig. 3. State transition diagram of Driver.

up, down, in_trip and *coordinate* messages (line 9, 13, 27, 31 and 42).

A Driver can keep multiple *dispatch request*s in its request queue. A newly received *dispatch request* is added to the queue (line 21) and processed in order. The maximum number of requests that the queue holds is determined by the parameter of back-to-back, that is a driver candidate selection algorithm (Section IV-A).

D. Matching Manager

Algorithm 4 defines the Matching Manager. The Matching Manager is initialized with a matching algorithm A and matching interval d. It performs matching between rider candidates and driver candidates based on the matching algorithm at every matching interval (line 8).

The matching process is organized into two phases as follows.

1) Selection of candidates for riders and drivers to be matched. In Algorithm 4, the Matching Manager performs the selection by updating the candidates, $rider\_cands$ and $driver\_cands$ (line 18, 24, 28 and 35).
2) Assignment of rider candidates to driver candidates (line 14).

A matching algorithm A provides the two phases. This organization facilitates the construction of various algorithms. Section IV describes what the current simulator implements as a matching algorithm A.

The second phase is just an assignment problem to minimize the sum of travel times. Algorithm 5 is the assignment process, that a matching algorithm A provides. It calculates travel times for all the pairs of the rider and driver candidates (line 7 to 15), and assigns the rider candidates to the driver candidates (line 16). For the assignment, a greedy algorithm works up to a moderate number of riders and drivers, and better algorithms are available, for example, the Hungarian algorithm.

We have choices for the first phase. The simplest algorithm selects only unmatched riders and drivers that have received

Algorithm 4 Matching Manager

```
Require: matching algorithm: $A$, interval: $d$
 1: procedure MATCHING_MANAGER($A$, $d$)                ▷ Constructor
 2:     $self.matching\_algorithm \leftarrow A$
 3:     $self.interval \leftarrow d$
 4:     $self.rider\_cands \leftarrow []$                 ▷ rider candidates
 5:     $self.driver\_cands \leftarrow []$                ▷ driver candidates
 6: end procedure
 7:
 8: loop
 9:     $self.do\_matching(self.rider\_cands, self.driver\_cands)$
10:     Sleep for $self.interval$
11: end loop
12:
13: procedure DO_MATCHING($rider\_cands$, $driver\_cands$)
14:     $pairs \leftarrow self.matching\_algorithm.assign$ ($rider\_cands$, $driver\_cands$)
15:     for all $(driver, rider) \in pairs$ do
16:         Send matching_notification($driver$) to $rider$
17:         Send dispatch_request($rider$) to $driver$
18:         Update $self.rider\_cands$ and $self.driver\_cands$ using $self.matching\_algorithm$
19:     end for
20: end procedure
21:
22: when upon receiving ride_request($x_1, y_1, x_2, y_2$) from rider $r$
23:     $req \leftarrow (r, x_1, y_1, x_2, y_2)$
24:     Update $self.rider\_cands$ by adding $req$
25: end when
26:
27: when upon receiving cancel_request from rider $r$
28:     Update $self.rider\_cands$ by removing $r$
29:     if $r$ was matched to driver $d$ then
30:         Send ride_cancel($r$) to $d$
31:     end if
32: end when
33:
34: when upon receiving status_update_* from driver $d$
35:     Update $self.driver\_cands$ using $self.matching\_algorithm$
36: end when
```

Algorithm 5 Assignment of rider candidates to driver candidates (line 14 of Algorithm 4)

```
Require: rider candidates selection algorithm: $select\_riders$ (in matching algorithm $A$), driver candidates selection algorithm: $select\_drivers$ (in $A$), travel time calculation function $f$ (in a Map),
Ensure: list of matched pairs (driver, rider)
 1: $rider\_cands \leftarrow$ get by $select\_riders$
 2: $driver\_cands \leftarrow$ get by $select\_drivers$
 3: $matched\_pairs \leftarrow assign(rider\_cands, driver\_cands)$
 4: return $matched\_pairs$
 5:
 6: function ASSIGN($rider\_cands, driver\_cands$)
 7:     $cost\_matrix \leftarrow$ [len($rider\_cands$)][len($driver\_cands$)]
 8:     for $i$ in range (len($rider\_cands$)) do
 9:         for $j$ in range (len($driver\_cands$)) do
10:             $r \leftarrow rider\_cands[i]$
11:             $d \leftarrow driver\_cands[j]$
12:             $travel\_time \leftarrow f(r.spawn\_x, r.spawn\_y, d.x, d.y)$
13:             $cost\_matrix[i][j] \leftarrow travel\_time$
14:         end for
15:     end for
16:     $matched\_pairs \leftarrow$ pairs that minimize the sum of costs
17:     return $matched\_pairs$
18: end function
```

no dispatch request. The algorithms utilizing back-to-back (Section IV-A) and reassignment (Section IV-B) select more riders and drivers aggressively.

E. Limitations of the model

1) Single region: The current model above assumes a ridesharing service at city scale and deals with the whole service area as the single region. It is necessary to limit the candidate riders and drivers based on their proximity to scale the service to continental scale. Grid systems, such as Uber's H3 [6], are available for the purpose, and our next steps include such an extension of the model.

2) No refusal by driver: In the current model, a driver automatically accepts all dispatch requests. In a real service, a driver can refuse a dispatch request to accept an upcoming better request or simply take a break. In the case of a small number of drivers, a driver has many options and such a refusal can worsen both rider and driver metrics such as rider waiting time and driver ride time.

3) No carpooling: A driver can carry only one rider at a moment. Depending the service, a user has the option of occupancy or carpooling. Carpooling would increase the total transport capacity and then may improve rider waiting time. It is part of our next steps to consider carpooling.

III. Simulator

Figure 4 shows the architecture of our simulator implemented along the model described in Section II. The reason we chose Java as the implementation language because of performance and portability. After reading a simulation scenario, it simulates the ridesharing service while using a routing system, such as OSRM. Then it writes out a statistical output and a simulation event log. The event log can be visualized with our visualizer and kepler.gl [7], an open-source software.

The simulator is event-driven, not time-stepped. The ridesharing simulation is an agent simulation and almost all state changes in the simulator only occur at the agent's events

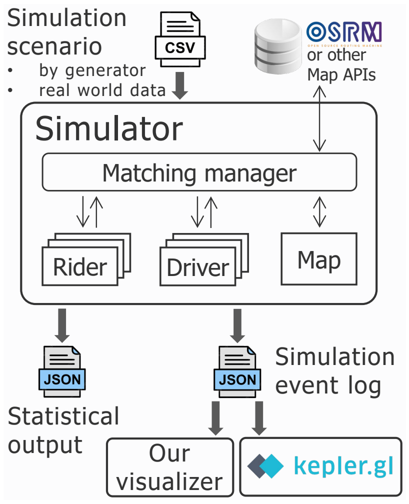


Fig. 4. Simulator architecture.

such as spawn, despawn, matching, pick up and drop off. Therefore, event-driven simulation is more time-effective. For this reason, number of simulators in various fields, such as peer-to-peer [8], [9] and blockchain [10], [11], [12], have adopted event-driven simulation.

The simulator accepts various rider and driver situations, road situations and matching algorithms. An input scenario describes the spawn time and other parameters of riders and drivers. A simulation reflects road situation by calling the travel function f given to the Map (Section II-A). We can compare matching algorithms by changing the algorithm A given to Matching manager (Section II-D).

A. Input: simulation scenario

A simulation scenario describes the behavior of all riders and drivers. In a scenario, a single entry is about a rider or a driver and includes spawn coordinate, spawn time and despawn time. A rider entry includes its destination additionally. Such a static scenario, not dynamically generated one, facilitates reproduction and fine control of the rider and driver situations.

There are several ways to prepare a scenario, including handwriting, using a generator, and converting from a real world trace. Our generator writes out a scenario along given settings, number of riders and drivers, temporal and geographic distribution of them and riders' destinations. We also convert our real world trace obtained in our service at Osaka city for analyses.

B. Utilizing routing systems

The simulator utilizes OSRM (Open Source Routing Machine) [5] to determine driver travel times. It means using OSRM as the travel function f given to the Map (Section II-A). We first selected OSRM because it works on OpenStreeMap that provides maps of extensive areas of the Earth.

OSRM is an open-source routing system implemented in C++. It runs as a server and provides an API, that takes an

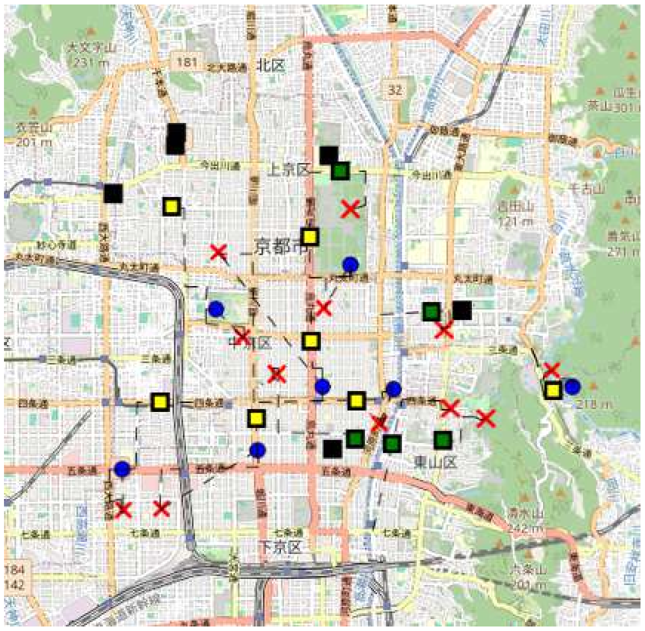


Fig. 5. Visualization using our visualizer.

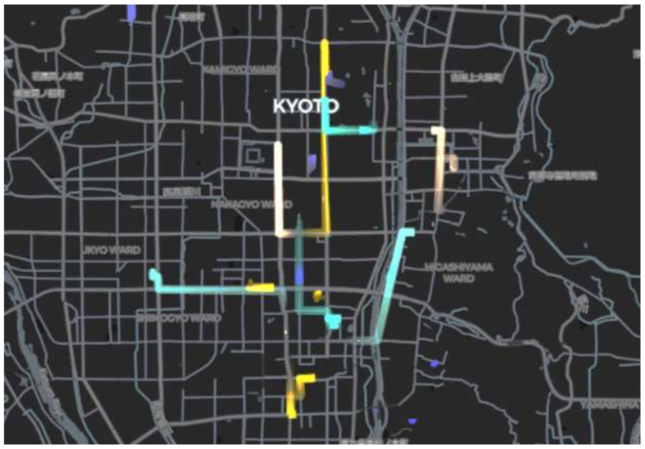


Fig. 6. Visualization using kepler.gl.

origin and a destination on the map, and returns routes and estimated travel times for the routes. We have two choices of where to place the server, the same server as the simulator or a different server from the simulator. There is a trade off. Zero network latency can be achieved by placing on the same server.

It is possible to utilize other map APIs, such as Google Maps, with some engineering work. Moreover, it is even possible not to use a routing system. The simulator provides a travel function f not using a routing system ("NO OSRM" in Section III-D), that returns the straight distance between the two points divided by speed. Such a simple travel time estimation is enough to compare matching algorithms relative to each other. For this reason, we did not select microscopic traffic simulators, such as SUMO (Simulation of Urban MObility) [13]. Microscopic simulations are not necessary for comparing

TABLE I
EXPERIMENTAL ENVIRONMENT FOR ALL THE EXPERIMENTS.

Processor	4 cores of Intel Xeon Platinum 8480+ (2.0 GHz)
Memory	16 GiB DDR5 SDRAM
	4 cores and 16 GiB are secured by Slurm, a job scheduler. The simulator and OSRM share the resources.
OS kernel	Linux 4.18.0-553.44.1.el8_10
Java runtime	OpenJDK 8 (version 1.8.0_442)
OSRM	Master branch as of October 24th, 2024 (After 5.27.1 and before 6.0RC1)

TABLE II
PARAMETERS FOR EXPERIMENTS ON EXECUTION TIME.

Matching algorithm	
Candidate selection	Only unmatched riders and drivers (Reass. level None and B2B level 0)
Assignment	Greedy algorithm
Matching interval	5 seconds
Travel time calc. function	NO OSRM and OSRM
	OSRM runs on the same server as the simulator.
Note that the followings are the same as Table III.	
Driver spawn time	Simulation start time
Driver despawn	None
Rider spawn time	Random
Rider despawn time	Random 0 $\sim$ 1200 sec after spawn
Map used by OSRM	Kyoto city, Japan
Rider spawn position	In a circle of radius 5 km
Rider destination	In a circle of radius 3 km centered at spawn position

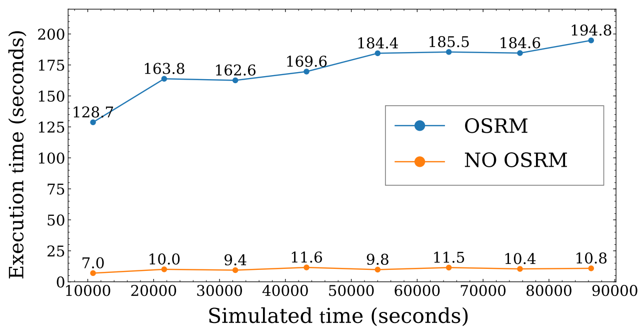


Fig. 7. Execution time vs. simulated time.

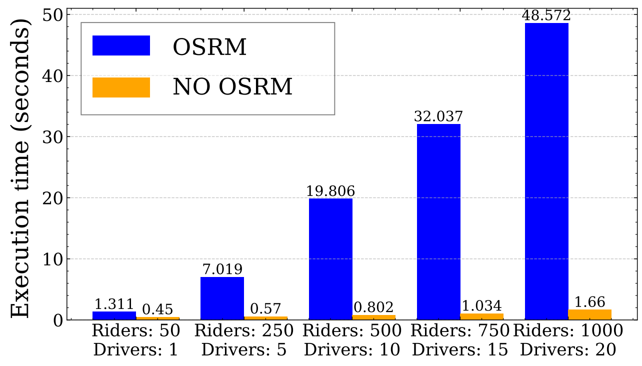


Fig. 8. Execution time vs. number of riders and drivers.

matching algorithms.

C. Output: statistical output and event log

The simulator writes out a statistical output and a simulation event log. The former is for analysis, and the latter is mainly for visualization. The results shown in the evaluation of matching algorithm (Section V) are based on the statistical output.

There are two ways for visualization, our visualizer and kepler.gl [7], an open-source software. Figure 5 shows a captured image of our visualizer, and Figure 6 shows one of kepler.gl. Visualization helps confirmation that a matching algorithm is working properly, not only demonstration.

A simulator as software is evaluated by metrics such as execution time and memory footprint.

The memory footprint increases with the size of the simulation, i.e., the number of riders and driver. Both a Rider and a Driver are Java objects. The size of data which an object holds is at most a few dozen bytes. 1 GiB of memory can hold 10 million objects, that is a enough large number for a ridesharing service.

We measured the simulator's execution times while varying simulated time and number of riders and drivers. Table I describes the experimental environment for all the experiments. Table II shows parameters for the measurement of execution times. "OSRM" means the use of OSRM as the routing system for the Map. "NO OSRM" means that OSRM is not used, and a travel time is the straight distance between the two points divided by speed.

D. Evaluation of the simulator

Figure 7 shows execution times for various simulated times from 10800 to 86400 seconds, that is from 3 to 24 hours. The number of riders and drivers is 1000 and 100 respectively. The execution times without OSRM is around 10 seconds even for 24 hours. The time does not increase linearly with the simulated time. It is an advantage of event-driven simulation. Using OSRM increased the execution time, but a single simulation was completed in 200 seconds even for 24 hours. The result also shows that OSRM calls, not the simulation itself, dominate the execution times.

Figure 8 shows execution times for various numbers of riders and drivers. The simulated time is 21600 seconds, that is 6 hours. The execution time increased along the number of riders and drivers. The increase is not exactly linear, but is is not far from linear either.

In summary, the simulated time itself does not affect the execution time because the simulator is event-driven. OSRM calls dominates the execution time. OSRM is called once for each pairs of riders and drivers in the assignment process (Section II-D and Algorithm 5). The number of the OSRM calls is then the product of the number of riders and drivers at that time. Because of it, as the number of riders and drivers increases, so does the execution time.

IV. MATCHING ALGORITHMS

A matching algorithm does its work in the two phases, candidate selection and assignment (Section II-D). The cur-

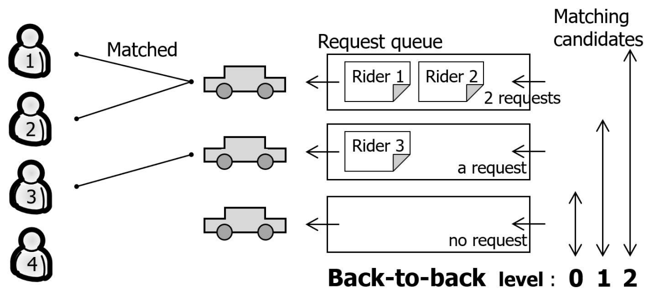


Fig. 9. Back-to-back.

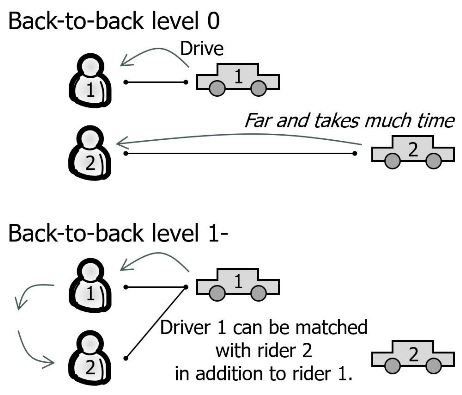


Fig. 10. Examples of back-to-back.

rent simulator implements back-to-back (Section IV-A) and reassignment (Section IV-B) for the candidate selection, and a greedy algorithm for the assignment.

A. Candidate driver selection

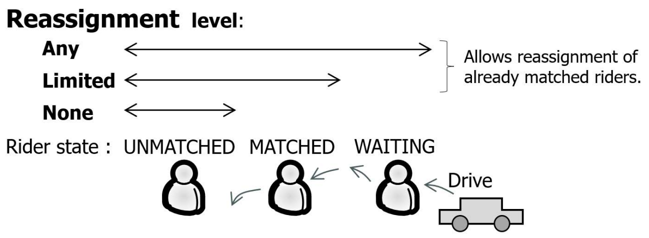
Back-to-back (B2B) determines how broad a range of drivers are matching candidates. B2B allows a driver who has already been matched with a rider to be matched with other riders. Figure 9 shows our definition of B2B. In the level N of B2B, a driver with N or fewer requests in its request queue is treated as a matching candidate.

As shown in Figure 10, the benefit of B2B is that it reduces the need to drive far to pick up a rider. It possibly reduces waiting time for riders.

In the simulator, each driver can have its own level of B2B. It is possible to apply B2B to part of the drivers as experimented in Section V-B.

B. Candidate rider selection

Reassignment determines how broad a range of riders are matching candidates. As shown in Figure 2, there five states that a rider can be in (Section II-B). Not only `UNMATCHED` but also `MATCHED` and `WAITING` states allows a rider to change the driver of the match. A driver change in the `MATCHED` and `WAITING` states means a change of the already matched driver. It is not only a change but also a cancellation if a driver is not assigned to the rider. It may cause the rider to have doubts about the reason. It should be carefully carried out and presented to the riders.



Reassignment level:
Any
Limited
None
Allows reassignment of already matched riders.
Rider state : UNMATCHED MATCHED WAITING
Drive

Fig. 11. Reassignment.

TABLE III
PARAMETERS FOR EXPERIMENTS OF MATCHING ALGORITHMS.

Number of riders	1000
Simulation time	21600 seconds = 6 hours
Matching algorithm	
Candidate selection	Depends on the experiment
Assignment	Greedy algorithm
Matching interval	5 seconds
Travel time calc. function	OSRM
	OSRM runs on the same server as the simulator
Note that the followings are the same as Table II.	
Driver spawn time	Simulation start time
Driver despawn	None
Rider spawn time	Random
Rider despawn time	Random 0 $\sim$ 1200 sec after spawn
Map used by OSRM	Kyoto city, Japan
Rider spawn position	In a circle of radius 5 km
Rider destination	In a circle of radius 3 km centered at spawn position

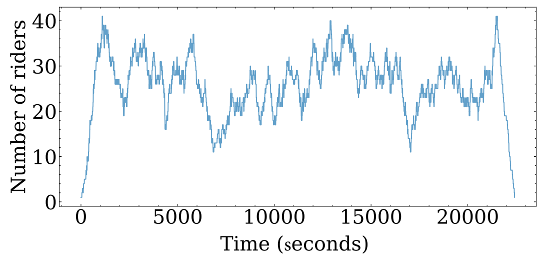


Fig. 12. Number of riders in the simulated 6 hours.

Figure 11 shows our definition of reassignment. There are three levels, None, Limited and Any. The first level, None, only treats riders in the `UNMATCHED` state as matching candidates. The second level, Limited, treats riders in the `MATCHED` state in addition. The maximum level, Any, treats riders in the `WAITING` state as well.

V. Evaluation of Matching Algorithms

We evaluate matching algorithms, that utilize different levels of the back-to-back (B2B) and the reassignment. The metrics of the evaluation are as follows.

1) Waiting time for riders
2) Total transport time for drivers
3) Number of travel time calculations

Waiting times have a direct impact on rider satisfaction. A short waiting time also reduces riders' cancellation, that is shown as the number of despawn in a simulation. A matching algorithm has the same effect as an additional driver input if

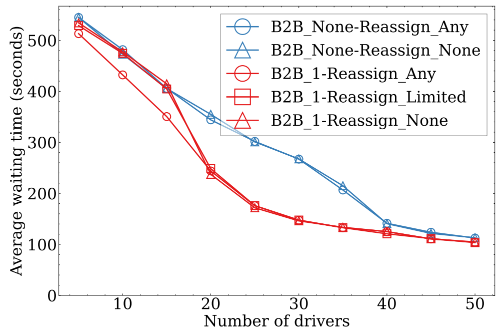


Fig. 13. Average waiting time for riders.

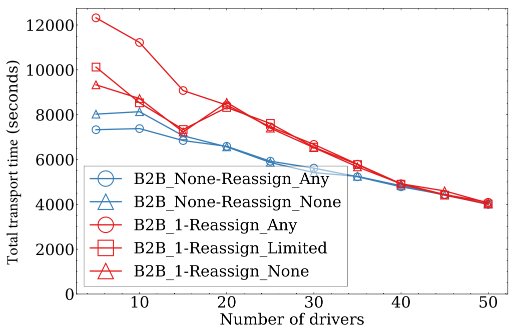


Fig. 14. Average total transport time for drivers.

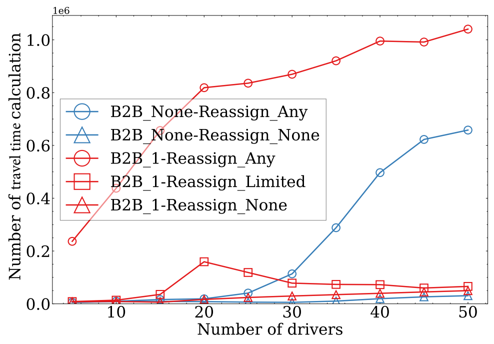


Fig. 15. Number of travel time calculations.

it reduces cancellation. Of course, fewer cancellations would simply increase income from the riders.

Total transport time influences driver income. Longer in the working hours is basically better.

Travel time calculations increase operator's expense if the operator uses a paid service, such as Google Maps API. A high-quality routing service improves matching accuracy, but its fee cannot be ignored.

Table III shows parameters for the following experiments. The number of riders is 1000. A rider is spawned at a random simulated time in the 6 hours and sends a ride request. Figure 12 shows the number of riders that are present at a given simulated time within the 6 hours. There are 15 to 40 riders are present for the most simulated time of a simulation. The number of drivers is from 5 to 50 in increments of 5. The experiments use OSRM.

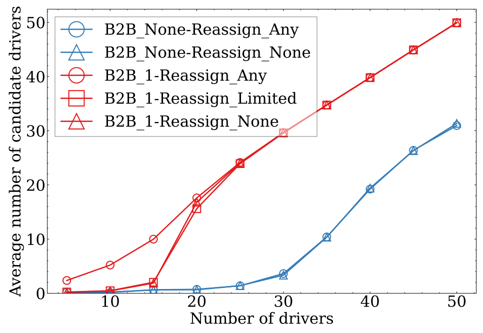


Fig. 16. Average number of candidate drivers.

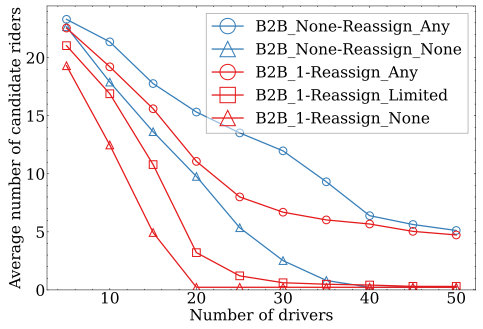


Fig. 17. Average number of candidate riders.

A. Impacts of B2B and reassignment

Experiments in this section measure the impact of B2B and reassignment in the above three metrics. The experiments target B2B levels 0 (no B2B) and 1, and all reassignment levels None, Limited and Any. Note that the number of combinations of B2B and reassignment levels is five, not six, which is two times three, because reassignment levels Limited and Any are the same for B2B level 0. Figure 13, 14 and 15 show the results. In the figures, "B2B_None" means B2B level 0 and "B2B_1" means level 1. Blue lines show the results of B2B level 0, and red lines show B2B level 1.

Figure 13 shows the average waiting time for riders. The more drivers, the shorter the waiting time naturally. The results differ depending on the degree of crowding.

- In the crowded situation, i.e. the number of driver is 15 or less, the level Any of reassignment shortened the waiting times.
  - Around 15% reduction in case of 15 drivers
- In the moderate situation, i.e. the number of driver is from 20 to 35, B2B shortened the waiting times significantly. Note that the number of drivers is comparable with the number of riders on the map shown in Figure 12.
  - Around 40% reduction in case of 25 drivers
- In the vacant situation, i.e. the number of driver is 40 or more, neither B2B nor reassignment affected much. B2B affects a little.

We can compare the five combinations not only with the same number of drivers (vertically) but also with the same waiting

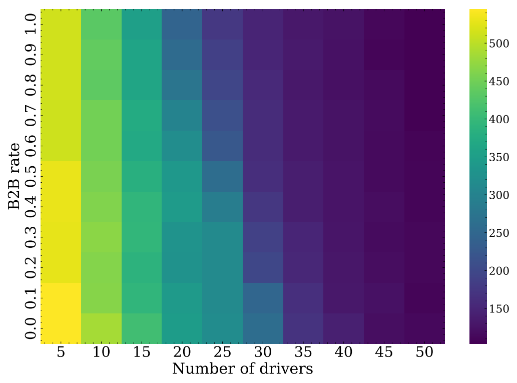


Fig. 18. Average waiting times.

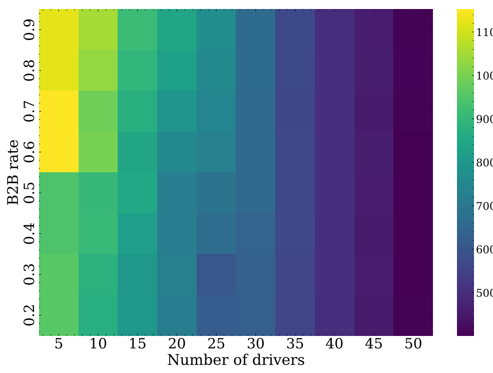


Fig. 19. Difference in total transport time between with and without B2B.

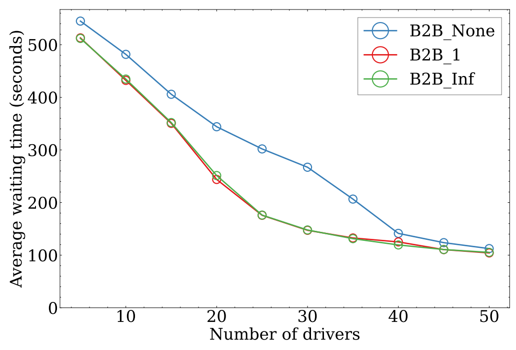


Fig. 20. Average waiting time with the level infinity of B2B.

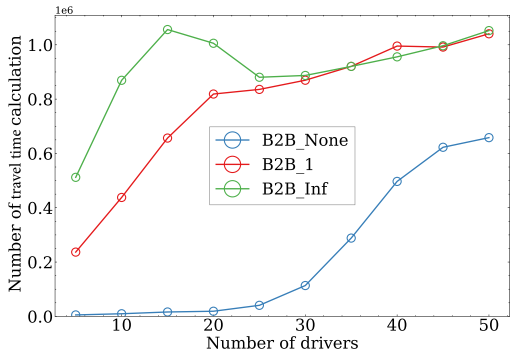


Fig. 21. Number of travel time calculations with the level infinity of B2B.

time (horizontally) on Figure 13. The waiting times with B2B and 20 drivers is comparable with the waiting times with no B2B and 30 or 35 drivers. The result shows that B2B had the same impact as increasing the number of drivers from 20 to 30 or 35. In summary, in the moderately crowded situation, B2B had much impacts in the waiting time. In the vacant situation, reassignment had a little impacts.

Figure 14 shows the average total transport time for drivers. Naturally, increasing the number of drivers reduced the total transport times.

- In the crowded situation, the level Any of reassignment increased the total transport time. B2B also had a little impact.
- In the moderate situation, B2B increased the total transport time.
- In the vacant situation, neither B2B nor reassignment affected much.

Figure 15 shows the number of travel time calculations. Reassignment increased the number of the calculation on its own. B2B had no impact on its own, but has an impact when combined with reassignment. Consistently, the level Any of reassignment showed the largest number of the calculations. Limited and None followed. It looks natural because the reassignment increases the number of candidate riders.

Figures 16 and 17 show the average number of matching candidates. Naturally, increasing the number of drivers increased candidate drivers and decreased candidate riders. It is also natural that B2B increased candidate drivers as shown in Figure 16, and reassignement increased candidate riders as shown in Figure 17.

B. Partial application of B2B

It is possible to apply B2B to part of the drivers as mentioned in Section IV-B. Figures 18 and 19 show the impact of partially-applied B2B. The B2B rate in the figures is the rate of drivers to whom B2B level 1 applies. The other drivers are in B2B level 0 (no B2B). The level of reassignment is Any.

Figure 18 shows the average waiting time for riders. The more drivers with B2B, the shorter the waiting time. The more upper, the darker in the figure.

Figure 19 shows the difference in the total transport times between drivers with B2B and those without. Driver with B2B consistently had longer transport times. The less drivers, the more crowded, the larger the difference. The more left, the lighter in the figure. The result shows that a driver can increases income by enabling B2B especially in a crowded situation.

C. Higher levels of B2B

The maximum level of B2B experimented so far is 1. Here we examine how higher levels of B2B affect the metrics.

Figures 20 and 21 show the results. In the figures, "B2B_Inf" means B2B level infinity. The level of reassignment is Any.

Figure 20 shows the average waiting time for riders. The level infinity did not show more benefit over the level 1. The benefit of B2B is that it reduces long drives to pick up a rider (Section IV-A). However back-to-back'ed riders must wait for the preceding riders to finish. We observed here that the benefit and the downside are already balanced with the level 1 of B2B.

Figure 21 shows the number of travel time calculations. The level infinity visibly increased the number in the crowded situation, i.e. the number of driver is 15 or less. Despite there is no benefit.

VI. Related work

Ota et al. [14] proposed a ridesharing simulator capable of large datasets and scenarios. Their goal is large-scale simulation, and their matching algorithm simply performs all-to-all matching based on shortest path lengths. In contrast, our goal is to experiment various matching algorithms as shown above. The evaluation of our simulator in Section III-D showed that the scalability of our simulator is sufficient for a city scale simulation. For larger scale, it is necessary to limit the matching candidates using a grid systems as mentioned in Section II-E1.

All the following related work utilizes a simulator to evaluate its proposal.

Lee et al. [15] proposed a driver assignment method based on Dijkstra's shortest path algorithm. In our simulator, OSRM serves the role of computing the shortest path.

Febbraro et al. [2] formulated ridesharing matching and demonstrated through simulation that rider waiting times can be improved by solving an optimization problem. Our contribution is a quantitative evaluation of matching candidate selection algorithms, B2B and reassignment, performed prior to optimization.

Maciejewski et al. [16] demonstrated the effectiveness of reassigning ride requests via simulation. Our contribution includes a comprehensive formulation of reassignment, and its relationship with B2B.

Jung et al. [3] investigated how carpooling between riders affects their waiting times. They compared a greedy method that assigns based on the shortest assignment distance, a method that minimizes the total travel time for riders, and a method that minimizes the total travel cost for riders, showing that the greedy method based on assignment distance results in the shortest rider waiting time. Our goal was to evaluate rider-driver matching algorithms prior to carpooling, that is part of our future work.

Martinez et al. [17] and Azevedo et al. [18] showed that the larger the number of drivers, the smaller the decrease in rider waiting times when a certain number of drivers are added. It is consistent with our results shown in Section V-A.

VII. Summary

This paper presented our model of ridesharing and the simulator developed based on the model. They satisfies the needs of us, i.e. a ridesharing operator, for experiments on matching algorithms.

We formulated techniques for rider-driver matching, back-to-back and reassignment on the model. The formulation enabled simulations. The simulation results showed that back-to-back reduced the average waiting time of riders by up to around 40% in the moderately crowded situation. Reassignment reduced it by up to around 15% in the crowded situation.

We will check and ensure that the simulation results are consistent with our real service. It is costly, as mentioned in Section I, but possible by comparing the metrics before and after the application of a technique such as back-to-back. It is part of future work as well as breaking the limitations described in Section II-E.

Code availability

All source code we developed for this study is available at https://github.com/shudolab/rideShareSim.

References

[1] Nader Masoud and R. Jayakrishnan. A real-time algorithm for rideshare matching problem with dynamic travel times. *Transportation Research Part B: Methodological*, 106:218–236, 2017.

[2] Angela Febbraro, Edoardo Gattorna, and Nicola Sacco. Optimization of dynamic ridesharing systems. In *16th Meeting of the EURO Working Group on Transportation*, pages 526–536, 2013.

[3] Jaeyoung Jung, R. Jayakrishnan, and J. Park. Dynamic shared-taxi dispatch algorithm. *Transportation Research Record*, 2359:36–45, 2013.

[4] Wenwen Zhang, Subhrajit Guhathakurta, Jason Fang, and Ge Zhang. Exploring the impact of shared autonomous vehicles on urban parking demand. *Transportation Research Part C: Emerging Technologies*, 71:19–34, 2016.

[5] Open Source Routing Machine (OSRM). Open source routing machine documentation. https://project-osrm.org/, 2025. Accessed on 2025-8-25.

[6] Uber Technologies Inc. H3: Uber's hexagonal hierarchical spatial index. https://eng.uber.com/h3, 2018. Accessed on 2025-8-25.

[7] kepler.gl. kepler.gl - powerful open source geospatial analysis tool. https://kepler.gl/, 2025. Accessed on 2025-8-25.

[8] Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi. Overlay Weaver: An overlay construction toolkit. *Computer Communications (Special Issue on Foundations of Peer-to-Peer Computing)*, 31(2):402–412, February 2008.

[9] Overlay Weaver: An overlay construction toolkit. http://overlayweaver.sf.net/. Accessed on 2025-8-25.

[10] Yusuke Aoki, Kai Otsuki, Takeshi Kaneko, Ryohei Banno, and Kazuyuki Shudo. SimBlock: A blockchain network simulator. In *Proc. IEEE INFOCOM 2019 Workshops (CryBlock 2019)*, April 2019.

[11] Ryohei Banno and Kazuyuki Shudo. Simulating a blockchain network with SimBlock. In *Proc. IEEE ICBC 2019*, pages 3–4, May 2019.

[12] SimBlock. https://dsg-titech.github.io/simblock/. Accessed on 2025-8-25.

[13] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic Traffic Simulation using SUMO. In *Proc. IEEE ITSC 2018*, 2018.

[14] Masashi Ota, Huy Vo, Claudio Silva, and Juliana Freire. Stars: Simulating taxi ride sharing at scale. *IEEE Transactions on Big Data*, 3(3):349–361, Sept. 2017.

[15] Der-Horng Lee, Huijuan Wang, Ruey Long Cheu, and Ching-Hoong Teo. Taxi dispatch system based on current demands and real-time traffic conditions. *Transportation Research Record*, 1882:193–200, 2004.

[16] Michal Maciejewski, Jose M. Salanova, Joschka Bischoff, and Marc Estrada. Large-scale microscopic simulation of taxi services. berlin and barcelona case studies. *Journal of Ambient Intelligence and Humanized Computing*, 7(3):385–393, 2016.

[17] Luis M. Martinez, Goncalo H.A. Correia, and Jose M. Viegas. An agent-based simulation model to assess the impacts of introducing a shared-taxi system. *Transportation Research Procedia*, 10:846–855, 2015.

[18] Carlos Lima Azevedo, Katarzyna Marczuk, Sébastien Raveau, Harold Soh, Christopher Zegras, and Moshe Ben-Akiva. Microsimulation of demand and supply of autonomous mobility on demand. *Transportation Research Record*, 2564:21–30, 2016.