Real-Time Driver-Request Assignment in Ridesourcing

Hao Wang, Xiaohui Bei

School of Physical and Mathematical Sciences, Nanyang Technological University
wang1242@e.ntu.edu.sg, xhbei@ntu.edu.sg

Abstract

Online on-demand ridesourcing service has played a huge role in transforming urban transportation. A central function in most on-demand ridesourcing platforms is to dynamically assign drivers to rider requests that could balance the request waiting times and the driver pick-up distances. To deal with the online nature of this problem, existing literature either divides the time horizon into short windows and applies a static offline assignment algorithm within each window or assumes a fully online setting that makes decisions for each request immediately upon its arrival. In this paper, we propose a more realistic model for the driver-request assignment that bridges the above two settings together. Our model allows the requests to wait after their arrival but assumes that they may leave at any time following a quitting function. Under this model, we design an efficient algorithm for assigning available drivers to requests in real-time. Our algorithm is able to incorporate future estimated driver arrivals into consideration and make strategic waiting and matching decisions that could balance the waiting time and pick-up distance of the assignment. We prove that our algorithm is optimal ex-ante in the single-request setting, and demonstrate its effectiveness in the general multi-request setting through experiments on both synthetic and real-world datasets.

Introduction

On-demand ridesourcing has become a pillar in urban transportation due to its efficiency and flexibility in connecting available drivers with on-demand riders. One of the essential components for a successful ridesourcing service is the real-time assignment of drivers to riding requests. However, the dynamic nature of on-demand ridesourcing brings several major challenges. More specifically, on a ridesourcing platform, drivers and riding requests are constantly arriving at unpredictable times. Upon the arrival of a riding request, its driver assignment needs to be computed in a very short time while considering the often conflicting objectives of minimizing request's waiting time, minimizing the driver's pick-up distance, as well as maximizing the number of served requests. The research community has only recently started to look at these problems, and many issues have not been addressed in satisfactory by existing works.

In the literature, there are mainly two approaches for solving the driver-request assignment problem. The first line of works (Cordeau and Laporte 2007; Bei and Zhang 2018) adopts a static and offline approach. That is, they focus on a single snapshot of the system and aims to find an optimal matching between all available drivers and the active requests in the snapshot. When dealing with real-time assignments, a common approach is then to divide the whole time horizon into small intervals (say one minute per interval) and apply the offline algorithm at the end of each interval (Anshelevich et al. 2013; Alonso-Mora et al. 2017; Lesmana, Zhang, and Bei 2019; Ke et al. 2020). With the information of all the drivers and requests at hand, the algorithm is usually able to find very efficient assignments through methods like maximum weight matching. However, in order to accumulate a sufficiently large pool of drivers and requests, the platform needs to set a long enough interval length, which will in turn incur a large *waiting cost* for the requests. In addition, because all assignments are being made at the end of an interval, requests that arrived at the beginning and at the end of an interval will experience a significantly different amount of waiting time. For example, a request arriving at the beginning of a one-minute interval needs to wait for a whole minute before it can be assigned, even if there is a nearby driver waiting at the very start. This will create significant inefficiency to the assignment, as well as a very unbalanced user experience for different riders, which will in turn harm the sustainability of the platform in the long run.

A second approach applies a fully online model that requires each request to be assigned (or rejected) *immediately* after its arrival. This approach has the advantage of reducing a request's waiting time to essentially zero. There is also a large body of literature on online matching from other domains, such as online bipartite matching (Karp, Vazirani, and Vazirani 1990; Mehta et al. 2007) and AdWords display (Aggarwal et al. 2011), from which the ridesourcing algorithm could draw inspirations. On the other hand, by completely removing the waiting time, the algorithm has to make haste and local decisions which will compromise the efficiency of the assignment, which we define as the distance between the request and the assigned driver in the assignment. In particular, a request that is assigned right upon its arrival loses the opportunity to be matched to closer drivers that might arrive in the near future.

From these two approaches, we can see a clear trade-off between the request waiting time and the assignment quality. In order to balance these two objectives, an ideal assignment algorithm should sit in the middle of these two approaches. That is, it should find the appropriate assignment time of a request that could strike a right balance between waiting for new drivers to emerge nearby and matching the request to existing drivers. This calls for a new driver-request assignment model that could capture the trade-off between request waiting time and assignment quality, together with an efficient assignment algorithm that could assign requests to the right drivers at the right time. These are the main objectives of this paper.

Our Contribution

In this paper, we propose a new driver-request real-time assignment model. In this model, both drivers and riding requests arrive in an online fashion, and the algorithm is allowed to make driver-request assignments at *any time*. To capture the waiting time and assignment quality trade-off, we assume the cost of an assignment consists of two components: the waiting cost, which is the time the request has waited before it gets assigned, and the matching cost, which is the pick-up distance between the request and its assigned driver. In addition, we will also assume that while waiting, a request might also quit and leave the platform at any time according to a quitting probability function. If the request leaves the platform unassigned, it will incur a quitting penalty. This assumption captures the impatient nature of the riding requests and makes the model more realistic. We also take a data-aware view and assume the algorithm knows the arrival distribution of the available drivers, which in practice can be estimated from historical data.

Based on this model, our second contribution is an efficient algorithm for assigning drivers to requests in real-time. At the heart of our algorithm, we want to answer the following question of a typical scenario: *When there is an active request and some drivers waiting, how long should the algorithm wait in the hope of a better (i.e. closer) driver arriving soon before assigning this request to the currently available driver?* To answer this question, we start from the simple case with only a single request and show how to compute the optimal waiting time thresholds when there are multiple types of drivers with different arrival rates. Next, we use the single-request algorithm as a building block to construct an efficient algorithm for the general case with multiple heterogeneous requests and drivers. Our algorithm is efficient and only requires a small polynomial update time for each arrival event. We also demonstrate the effectiveness of our algorithm through experiments on both synthetic and real-world datasets.

Related Works

Driver-request assignment in ridesourcing has been studied extensively in the literature across multiple disciplines. As discussed above, most works can be categorized into the offline static assignment setting and the online dynamic assignment setting.

On the offline front, in operation research, the problem is under the name of *dial-a-ride problem (DARP)* and has been studied in several works (Colorni and Righini 2001; Coslovich, Pesenti, and Ukovich 2006; Cordeau and Laporte 2007). In computer science, the offline assignment problem has been considered in the context of ride-sharing (Bei and Zhang 2018), efficiency and fairness balance (Lesmana, Zhang, and Bei 2019), route optimization (Alonso-Mora et al. 2017). Many of these works also use the batch approach (i.e. divide the time horizon into short intervals) to convert the offline algorithm to solve the real-time assignment problem.

On the online front, several works have investigated the online ridesourcing assignment problem (Lee et al. 2004; Caramia et al. 2001; Bertsimas, Jaillet, and Martin 2019; Ma, Zheng, and Wolfson 2013; Xu et al. 2018; Miao et al. 2016; Dickerson et al. 2018b,a; Nanda et al. 2020). This problem is also closely related to online bipartite matching (Karp, Vazirani, and Vazirani 1990; Mehta et al. 2007; Aggarwal et al. 2011) in which one side of the vertices arrive online. All of these works require the request to be assigned immediately upon its arrival. That is, there is no waiting time involved.

There are also works that allow vertices to quit during the matching process. Huang et al. (2018, 2019) study an online general graph matching model in which every arrival vertex has a fixed deadline to be matched before it leaves the system. Collina et al. (2020); Aouad and Saritaç (2020) consider a general graph matching problem in which vertices arrive and depart following given processes. A major difference between these works and ours is they do not consider request waiting time as part of the matching cost.

Ke et al. (2020); Xu et al. (2018); Feng, Gluzman, and Dai (2021); Lowalekar, Varakantham, and Jaillet (2021) use reinforcement learning to solve different ridesourcing problems. (Ke et al. 2020) consider a model which is similar to our settings. However, their model still uses fixed time intervals and can only make matches at the end of each interval, while our model focuses on the real-time assignment.

Another related problem is the *min-cost perfect matching with delays (MCMD)* model studied in (Emek, Kutten, and Wattenhofer 2016; Azar, Chiplunkar, and Kaplan 2017; Azar and Fanani 2020; Ashlagi et al. 2017). Similar to our model, these works consider both waiting cost and matching cost as part of their objectives. The main difference is that in our model, only the request waiting time is counted while they consider the waiting cost of all vertices. They also do not allow vertices leaving the system voluntarily, which is different from our assumption.

Finally, queueing model is another related model that has been used for ridesourcing assignment problems before (Zhang and Pavone 2016; Zukerman 2013; Banerjee, Johari, and Riquelme 2015).

Model

We consider a bipartite graph $G_0 = (\mathcal{R}, \mathcal{D}, E)$ that is known to the algorithm. Here $\mathcal{R} = \{r | 1 \leq r \leq N\}$ and $\mathcal{D} = \{d | 1 \leq d \leq M\}$ are the type spaces of requests and drivers, respectively, with $|\mathcal{R}| = N$ and $|\mathcal{D}| = M$. E denotes the set of allowed matches between requests and drivers. In other words, a request of type r can be matched to a driver of

type d if $(r, d) \in E$. We do not put any restrictions on the structure of the graph and allow E to encode any physical or performance-related constraints. In practice, usually requests can only be matched to nearby drivers due to pick-up distance constraints. This means G_0 is usually a sparse graph. Each edge $(r, d) \in E$ is also associated with a cost c_{rd}. This cost represents the time for the driver to pick up the request. We consider an infinite time horizon starting from time 0, and a fully online setting where vertices from both sides arrive at different times.

Driver and Request Arrival. For each driver type $d \in \mathcal{D}$, the arrival of a driver of this type is independent of arrivals of other types and follows a known random process. In this paper, we limit our focus to the *Poisson process* with process rate λ_d. This is a standard assumption for driver arrivals that has been considered in many works (Collina et al. 2020; Aouad and Saritaç 2020). After its arrival, the driver will stay in the system until it is matched.

For each request type $r \in R$, we do not make any assumptions on its arriving process and allow it to be arbitrary. However, when a request of type r arrives, this request will only be available for a period of time before it quits and leaves the system. The length of time t that this request stays in the system follows a quitting distribution $F(t)$. More specifically, if a request arrives at time t_0, then $F(t)$ denote the probability that this request quits before time $t_0 + t$. Let $f(t) = \frac{dF(t)}{dt}$ be the probability density function of $F(t)$. We make a natural assumption that $F(t)$ is continuous and $f(t)$ is non-decreasing. That is, the requests are more likely to quit the longer they have been waiting.

Matching and Matching Cost. Our goal is to design an online algorithm that matches the requests and drivers in real time as they arrive. Note that our setting differs from the traditional online matching setting in that when a request or a drive arrives, we are not required to immediately find a match for it. Instead, we are allowed to match a request and a driver at any time as long as they are both still in the system.

For every request i arrived in the system, it will either be matched by the algorithm to some driver j at some time t, or it will quit at some time t_i^Q unmatched. In either case, this request will incur two types of costs: a *waiting cost* and a *matching cost*.

- The waiting cost is the waiting time of a request before it is matched, or it quits. That is, if request i arrives at time t_i and is matched at time t, then the waiting cost is $t - t_i$. If the request i quits at time t_i^Q without being matched to any drivers, the waiting cost is $t_i^Q - t_i$.
- The matching cost is c_{rd} when the request i (of type r) is matched to some driver j of type d. If the request i quits without being matched to any drivers, it will incur a fixed matching cost of c_r^Q. We assume $c_r^Q > c_{rd}$ for all $(r, d) \in E$.

The cost of a request is the sum of the waiting cost and the matching cost. The goal of the algorithm is to minimize the total cost of the matching, which is the sum of the costs of all requests.

Single Request

We start off by considering a simplified model with the following assumption.

Assumption 0.1. *There is a single request of type r arriving at time 0 in the whole time horizon.*

The reason for starting from this simplified setting is twofold: first, it can provide us useful intuitions about what should an optimal matching strategy look like, even in the general case; second, the solution of this simple case can also be used as a building block for the algorithms for the general, multi-request case. In the following we will also call this request r when the context is clear.

Assume there are M types of drivers, and type i driver has a matching cost $c_{ri} = b_i$. We assume without loss of generality that $b_1 < b_2 < \cdots < b_M < c_q$, where c_q is the quit cost of request r. We also assume the arrival rate of type i driver is λ_i, and we denote the vector $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_M)$.

First, observe that at any time when there are multiple drivers of different types waiting, only the driver with the smallest matching cost should be considered. Therefore, throughout the process, we only needs to maintain two pieces of information: the current time t and the best driver type m that is currently available. We denote this state as $s(t, m)$.

At state $s(t, m)$, intuitively, the algorithm faces two choices: wait or match. If the time is still early or the available driver m is far away, we may want to wait a bit to see if a closer driver can arrive soon (unless $m = 1$ is already the closest driver type, in which case the algorithm should match r to this driver immediately). On the other hand, as time grows, the probability of the request quitting will increase, until it becomes large enough such that the chance of receiving a quitting cost c_q will outweigh the potential benefits of waiting for a better driver. Then the algorithm should simply match r to the type m driver.

Based on this intuition, we can see that the optimal algorithm in this setting should be in the form of a parameterized *threshold algorithm* SINGLE-REQ$_r(\boldsymbol{\lambda}, \boldsymbol{T} = (T_1, T_2, \ldots, T_{M+1}))$ described as below.[1] Here T_m is the threshold waiting time for state $s(t, m)$, such that when $t < T_m$, the algorithm should wait, and when the time reaches T_m with no better driver arrived, the algorithm would match the request to this type m driver. Note that if there is better driver $m' < m$ arrived at time t' before T_m, the algorithm does not necessarily have to match r to this driver m' immediately. Rather, it will transit to a new state $s(t', m')$ and use the new threshold waiting time $T_{m'}$ to decide whether to continue waiting or not. [2] The pseudocode can be found in Algorithm 1.

The remaining question boils down to how to choose the threshold waiting times $(T_1, \ldots, T_M)$ that is optimal for the algorithm.

[1] Without loss of generality, we only focus on deterministic algorithms here.

[2] For notational convenience, we define a new type $M + 1$ driver with $c_{r(M+1)} = \infty$ to represent the state of no available drivers. Clearly we would have $T_{M+1} = \infty$ because without any available drivers, the request has no other options except waiting.

Algorithm 1: Single Request and Multi Driver Types SINGLE-REQ$_r(\boldsymbol{\lambda}, \boldsymbol{T})$

Input: a single request r arrives at time 0;
M driver types with arrival rates $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_M)$;
Threshold $\boldsymbol{T} = (T_1 = 0, T_2, T_3, \ldots, T_M, T_{M+1} = \infty)$.

$m \leftarrow M + 1$ // m represents the best available driver type so far
while time t grows continuously starting from 0 **do**
  if A type d_i driver arrives with $i < m$ **then**
    $m \leftarrow i$
  end if
  if $t \geq T_m$ **then**
    match request r to the d_m driver
    return
  end if
end while

Finding the optimal threshold times $\boldsymbol{T}^*$. Next we provide a characterization of the optimal thresholds in algorithm SINGLE-REQ for the case of a single request and multiple driver types.

Theorem 1. *With a single request r and M types of drivers, the optimal algorithm is the threshold algorithm* SINGLE-REQ$_r(\boldsymbol{\lambda}, \boldsymbol{T} = (T_1^*, \ldots, T_{M+1}^*))$*, where T_j^* is the smallest nonnegative value satisfies*

$$q(T_j^*) \geq \frac{\sum_{i=1}^{j-1} \lambda_i (b_j - b_i) - 1}{c_q - b_j},$$

where $q(t) = \frac{f(t)}{1-F(t)}$ is the probability density function that the request quits conditioning on the request still being alive at time t. If no such T_j^* exists, then we set $T_j^* = \infty$ which means the algorithm will always wait no matter how long the request has waited.

Proof. Consider the scenario in which j is currently the best available driver type, and we are at time $t = T_j^*$. Consider the following two options.

- **Option (1)**: assign request r to currently the best driver, which will generate a cost of $b_j + t$.
- **Option (2)**: wait for a short time Δ, match r to any driver with type $i < j$ that arrives during time $(t, t + \Delta)$, and assign r to the type j driver at time $t + \Delta$ otherwise. For this option, there are j events could happen in the $[t, t + \Delta]$ time interval: the request quits (with probability $\Delta q(t) + o(\Delta)$) or a type $d_i (i < j)$ driver arrives (with probability $\Delta\lambda_i + o(\Delta)$). Again we ignore the probability that two or more such events happen within this short time interval. If a driver of type $i < j$ arrives, the algorithm will assign r to this driver with a total cost of $b_i + t + O(\Delta)$. In summary, the expected cost of this option is

$$t + \Delta \sum_{i<j} \lambda_i b_i + \Delta q(t) c_q + \left(1 - \Delta \sum_{i<j} \lambda_i - \Delta q(t)\right)(\Delta + b_j) + o(\Delta).$$

It is not hard to see that the optimal threshold T_j^* should be the smallest time t where the cost of option (1) becomes smaller than the expected cost of option (2). This is because the request's quitting probability $f(t)$ is non-decreasing. Therefore, if matching d_j at time t is better than matching d_j at time $t + \Delta$, it should also be better than matching d_j at any time afterwards. By letting Δ approach 0, we have that T_j^* is the smallest value that satisfies

$$q(T_j^*) \geq \frac{\sum_{i=1}^{j-1} \lambda_i (b_j - b_i) - 1}{c_q - b_j}.$$

□

In the following we give an intuitive explanation of this theorem.

- $c_q - b_j$ is the cost difference between request r quitting and matching to a type d_j driver. $q(t)$ is the probability density of quitting at time t. This means $q(t)(c_q - b_j)$ is the expected increase of cost if we choose to wait (for a unit time period) but r quits during the period. Waiting will also incur an additional unit of waiting cost. Overall, compare to matching at time t, the downside of waiting is $q(t)(c_q - b_j) + 1$.
- $b_j - b_i$ is the cost difference between the request matching to a type d_j driver and matching to a type d_i driver. λ_i is the probability density of the arriving distribution of type d_i driver. Therefore, $\lambda_i(b_j - b_i)$ is the expected *decrease* of cost if we choose to wait and a type d_j driver arrives during the period.

When $t < T_j^*$, we have $q(t)(c_q - b_j) + 1 < \sum_{i<j} \lambda_i (b_j - b_i)$. In other words, the expected downside of waiting is less than its upside. Therefore, we should continue waiting until t reaches T_j^*.

Extensions. In our model, we assume the waiting cost is exactly the waiting time. We can generalize it to the setting that the waiting cost is a function $h(t)$ of waiting time t. Following a similar analysis, when $h(t)$ is a non-convex function, one can show that the optimal algorithm is still SINGLE-REQ, and the threshold time T_j^* is the smallest nonnegative value satisfying

$$q(T_j^*) \geq \frac{\sum_{i=1}^{j-1} \lambda_i (b_j - b_i) - h'(t)}{c_q - b_j},$$

where $h'(t)$ is the derivative of $h(t)$.

Multiple Requests

In this section we investigate the general case with multiple heterogeneous requests and drivers. Following the single-request case in Section , we can model this problem as a *Continuous Markov Decision Process (CMDP)*. Unfortunately, similar to many other real-world problems, this CMDP problem suffers from the *curse of dimensionality* and *curse of modeling*. That is, the CMDP has multi-dimension and continuous state space, with very complicated transition probabilities and reward structure. As a result, traditional policy or value iteration methods are infeasible, and we have to rely

on *approximate* solutions. In the following, we will show a simple and efficient approximation approach that takes into consideration the specific problem structure and also makes use of the single-request optimal solution that we obtained in Section .

Main idea. At any time t, assuming that there are currently n available requests and m available drivers, our main idea is that we first "virtually" match each available request i with at most one available driver d^i, and then *decompose* our problem into n subproblems, where each subproblem contains only a single request with zero or one available driver *exclusively* waiting for this request. Note that any new driver arrives after time t can be assigned to at most one of these subproblems, which means these n subproblems also need to "share" the arrival rate of each driver type. Thanks to the decomposition property of the Poisson arrival process, we can "decompose" a Poisson process with rate λ into n independent Poisson subprocesses with rates $\lambda_i (1 \le i \le n)$ as long as it satisfies $\sum_i \lambda_i = \lambda$.[3] This allows us to conveniently distribute both the available drivers and the driver arrival rates to each request, therefore reducing our problem to n different single-request problems, for which we know the exact optimal strategy from Section .

This idea leads us to the following algorithm MULTI-REQ (Algorithm 2) for solving the general heterogeneous case.

Algorithm 2: Heterogeneous Requests and Driver Types MULTI-REQ($\boldsymbol{\lambda}$)

Input: M driver types with arrival rates $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_M)$;

while time t grows continuously starting from 0 **do**
  $R \leftarrow$ currently available requests
  $D \leftarrow$ currently available drivers
  if a new request i_{new} arrives **then**
    create a new SINGLE-REQ$_{i_{\text{new}}}$ starting from t
  end if
  if a request $i \in R$ quits **then**
    terminate SINGLE-REQ$_i$
  end if
  if any arrival or quit event happens **then**
    recompute $\{\boldsymbol{\lambda}^i\}$ and $\{d^i\}, \forall i \in R$
    recompute thresholds $\boldsymbol{T}^i$ from Thm 1, $\forall i \in R$
    update $\{\boldsymbol{\lambda}^i\}, \{d^i\}$, and $\{\boldsymbol{T}^i\}$ in SINGLE-REQ$_i(\boldsymbol{\lambda}^i, \boldsymbol{T}^i, d^i), \forall i \in R$ [4]
  end if
  while there exists request i with $t \ge T^i_{d^i}$ in SINGLE-REQ$_i$ **do**
    pick i with the smallest $T^i_{d^i}$ among all such requests
    match request i to the corresponding driver d^i
    terminate SINGLE-REQ$_i$
    recompute and update $\{\boldsymbol{\lambda}^i\}$ and $\{\boldsymbol{T}^i\}, \forall i \in R$
  end while
end while

[3] This can be done by assigning each arrival to subprocess i with probability λ_i/λ.

[4] We make two adjustments to the single-request algorithm SINGLE-REQ$_i(\boldsymbol{\lambda}^i, \boldsymbol{T}^i)$ from Section . First, we allow the single request i to arrive at any time t (not necessarily 0). It is easy to check that all results from Section still holds if we shift all threshold times by t. Second, we add a new parameter d^i to SINGLE-REQ$_i$ which simply denotes the driver that is already available to i from the starting time.

Given this algorithm framework, we have two important tasks left: how to allocate the available drivers in D and the driver arrival rates $\boldsymbol{\lambda}$ to these n requests in R to form the n subproblems. This task can be characterized by an optimization problem formulation with the following notations:

- $x_{ij} \in \{0,1\} (i \in R, j \in D)$: $x_{ij} = 1$ means we match request i to driver j. We also introduce a "null" driver j_0 with type $d_{j_0} = M + 1$, and use $x_{ij_0} = 1$ to denote that request i is unmatched. We denote $D^+ = D \cup \{j_0\}$.
- $U^{\boldsymbol{\beta}}_{rd}(t) (r \in \mathcal{R}, d \in \mathcal{D})$: the optimal expected cost function of the subproblem with a type r request that has waited for t time, an available driver of type d, and arrival rates vector $\boldsymbol{\beta}$.

$$
\begin{aligned}
P_0 : \min_{x_{ij}, \boldsymbol{\lambda}^i} \ & \sum_{\substack{i \in R \\ j \in D^+}} x_{ij} U^{\boldsymbol{\lambda}^i}_{r_i d_j}(t - t_i) \\
\textbf{s.t.} \ & \sum_{j \in D^+} x_{ij} = 1 && \forall i \in R \\
& \sum_{i \in R} x_{ij} \le 1 && \forall j \in D \\
& \sum_{i \in R} \boldsymbol{\lambda}^i = \boldsymbol{\lambda} \\
& x_{ij} \in \{0,1\} && \forall i \in R, j \in D^+ \\
& \boldsymbol{\lambda}^i \ge \mathbf{0} && \forall i \in R.
\end{aligned}
$$

$$
\begin{aligned}
P_1 : \min_{x_{ij}} \ & \sum_{\substack{i \in R \\ j \in D^+}} x_{ij} \widetilde{U}_{r_i d_j} \\
\textbf{s.t.} \ & \sum_{j \in D^+} x_{ij} = 1 && \forall i \in R \\
& \sum_{i \in R} x_{ij} \le 1 && \forall j \in D \\
& x_{ij} \in \{0,1\} && \forall i \in R, j \in D^+.
\end{aligned}
$$

$$
\begin{aligned}
P_2 : \min_{\boldsymbol{\lambda}^i} \ & \sum_{i \in R} \overline{U}_{r_i d^i}(\boldsymbol{\lambda}^i) \\
\textbf{s.t.} \ & \sum_{i \in R} \boldsymbol{\lambda}^i = \boldsymbol{\lambda} \\
& \boldsymbol{\lambda}^i \ge \mathbf{0} && \forall i \in R.
\end{aligned}
$$

Let P_0 denote the optimization problem. Unfortunately, we cannot solve P_0 directly because $U^{\boldsymbol{\lambda}^i}_{r_i d_j}(t - t_i)$ can be very

complex and does not have an explicit form. It also depends on the time t which means we need to recalculate this value at every possible time. Therefore, we have to find approximate solutions of this optimization problem.

We apply a two-stage approximation approach. First, we use a fix value $\widetilde{U}_{rd}$, which will be discussed later in details, as an approximation of all $U^{\boldsymbol{\beta}}_{rd}(t)$ with different $\boldsymbol{\beta}$ and t, and use these values to compute the best request-driver matching $\boldsymbol{x}^*$. We denote this first-stage optimization problem as P_1.

Next, given the matching $\boldsymbol{x}^*$, for each $i \in R$, there is only one driver d^i with $x_{id^i} = 1$. We then use another *linear* approximation $\overline{U}_{rd}(\boldsymbol{\beta})$, which also will be discussed later, to approximate $U^{\boldsymbol{\beta}}_{rd}(t)$ (note that this function has the arrival rates as its parameter but still ignores the information of time t), and compute a rate allocation based on these values. We denote this second-stage optimization problem as P_2.

Cost Function Approximation

In this section, we discuss in details how to design $\widetilde{U}_{rd}$ in P_1 and $\overline{U}_{rd}(\boldsymbol{\beta})$ in P_2. We want our design to satisfy two properties: (1) they are reasonable approximations of the actual cost function $U^{\boldsymbol{\beta}}_{rd}(t)$, and (2) they can be computed and maintained efficiently.

We start with $\overline{U}_{rd}(\boldsymbol{\beta})$. Fix any request type r and driver type d. Note that $U^{\boldsymbol{\beta}}_{rd}(t)$ describes the scenario with a single request of type r and a single available driver of type d. We can use Theorem 1 to compute the optimal time threshold $T^*_{rd}(\boldsymbol{\beta})$. However, the expected cost of the optimal algorithm does not have a simple closed-form and may be affected by the value of $\boldsymbol{\beta}$ and t in a complicated way. Therefore, we need to find simplifications to this single-request problem such that the expected can be computed and approximated more easily.

We will make the following simplifications. First, because the information of the request waiting time is not in the approximation, we will assume $t = 0$. That is, the request just arrives. Second, we will assume a simple quitting function for this request: it will stay in the system until time $T^*_{rd}(\boldsymbol{\beta})$ and quit right after. Third, we also assume a simple strategy: whenever a driver better than d arrives before time $T^*_{rd}(\boldsymbol{\beta})$, match it to the request immediately. If there are no such drivers, match the request to the only available driver at time $T^*_{rd}(\boldsymbol{\beta})$. Let $U'_{rd}(\boldsymbol{\beta})$ be the expected cost function with all these simplifications. We will find the explicit solution of $U'_{rd}(\boldsymbol{\beta})$, and then approximate it with a linear function $\overline{U}_{rd}(\boldsymbol{\beta})$ with arrival rates $\boldsymbol{\beta}$ as the function parameter.

We denote

$$\beta_{<d} = \sum_{d':c_{rd'}<c_{rd}} \beta_{d'} \quad \text{and} \quad c^{\boldsymbol{\beta}}_{<d} = \frac{\sum_{d':c_{rd'}<c_{rd}} \beta_{d'} c_{rd'}}{\sum_{d':c_{rd'}<c_{rd}} \beta_{d'}}.$$

Let $C(x)$ denote the algorithm cost when the first driver better than d arrives at time x. According to our simplified strategy, the algorithm will match this driver to the request right away. Because each type d driver arrives following a Poisson process with rate β_d, by the additive property of Poisson processes, we know x also follows a Poisson process with parameter $\beta_{<d}$, and conditioning on the arrival, the probability that this driver is of type d' is $\frac{\beta_{d'}}{\beta_{<d}}$. This gives us

$$C(x) = \begin{cases} c^{\boldsymbol{\beta}}_{<d} + x, & x \le T^*_{rd}(\boldsymbol{\beta}) \\ c_{rd} + T^*_{rd}(\boldsymbol{\beta}), & x > T^*_{rd}(\boldsymbol{\beta}) \end{cases}$$

From here we can then derive the expected cost function

$$\begin{aligned} U'_{rd}(\boldsymbol{\beta}) &= \int_0^\infty \beta_{<d} \exp\left(-\beta_{<d} x\right) C(x) dx \\ &= (c_{rd} - \widetilde{c}^{\boldsymbol{\beta}}_{<d}) \exp\left(-\beta_{<d} T^*_{rd}(\boldsymbol{\beta})\right) + \widetilde{c}^{\boldsymbol{\beta}}_{<d} \end{aligned}$$

where $\widetilde{c}^{\boldsymbol{\beta}}_{<d} = c^{\boldsymbol{\beta}}_{<d} + \frac{1}{\beta_{<d}}$.

Having obtained $U'_{rd}(\boldsymbol{\beta})$, our final task is to approximate it with a linear function. It is easy to see that $U'_{rd}(\mathbf{0}) = c_{rd}$. We will use the linear function $\overline{U}_{rd}(\boldsymbol{\beta})$ that goes through $(\mathbf{0}, c_{rd})$ and $(\boldsymbol{\lambda}, U'_{rd}(\boldsymbol{\lambda}))$ to approximate U'.

For $\widetilde{U}_{rd}$ in P_1, because it lacks the arrival rates information, we use the average value of this linear function between $(0, c_{rd})$ and $(\lambda_{<d}, U_d(\boldsymbol{\lambda}))$, that is, $\frac{1}{2}(U_d(\boldsymbol{\lambda}) + c_{rd})$, as the approximation.

Our final construction of $\widetilde{U}$ and $\overline{U}$ is summarized below.

Approximations in P_1 and P_2

- In P_1, we use:

$$\widetilde{U}_{rd} = \frac{1}{2}(U'_{rd}(\mathbf{0}) + U'_{rd}(\boldsymbol{\lambda})) = \frac{1}{2}(c_{rd} + U'_{rd}(\boldsymbol{\lambda})).$$

- In P_2, we use:

$$\overline{U}_{rd}(\boldsymbol{\beta}) = \frac{U'_{rd}(\boldsymbol{\lambda}) - c_{rd}}{\lambda_{<d}} \beta_{<d} + c_{rd}.$$

Implementation

We now discuss the implementation of the multiple-request algorithm MULTI-REQ. Although the algorithm describes a continuous time process, during which $\{\boldsymbol{\lambda}^i\}$, $\{d^i\}$, and $\{\boldsymbol{T}^i\}$ needs to be recomputed after every relevant event, we can implement it efficiently. In the following we show that using the approximation we just obtained, one can implement the algorithm in a discrete and very efficient way, such that it can be applied in practice in a reasonable scale.

Because $\{\boldsymbol{\lambda}^i\}$ and $\{d^i\}$ are derived from the optimization problem P_1 and P_2, we first discuss how to solve and maintain the optimal solutions of these two problems. In the following we let n to be the maximum total number of request and drivers in the system at any time, and let N denote the number of total arrivals. We will also analyze the time complexity of each step as a function of n and the time complexity of MULTI-REQ by n and N.

Solving and maintaining P_1. Note that P_1 is a standard maximum weight matching problem[5] whose optimal solution

[5]Technically it is a minimum cost perfect matching problem, though it can be easily converted to a maximum weight matching problem by replacing the cost c_{rd} of each edge by $M - c_{rd}$ with some large constant M.

needs to be maintained throughout the time horizon. Note that both P_1 and P_2 do not involve the current time t as a parameter. This means when time t grows without any events happened, the optimal matching will remain unchanged. Next we discuss how to deal with different events.

There are several events that will happen:

- When the time t hits $T^i_{d_i}$ in the subproblem SINGLE-REQ$_i$ for some request i: this means we need to assign request i to its matching driver d_i in the optimal matching at this time. Both request i and driver d_i will be removed from the graph. However, it is easy to see that the remaining matching is still a maximum weight matching in the remaining graph. Therefore, we don't need to apply any additional updates.
- A new driver or request arrives: we add a new vertex to the bipartite graph and find the smallest negative cycle that contains this vertex (if there is any), and update the matching via the augmenting path according to this cycle.
- A request i quits: we remove the corresponding vertex from the graph, and treat its matched driver as a new arriving driver to the graph.

Regarding the implementation of this update, we maintain a residual graph of the optimal solution and the negative cycle can be detected using the Bellman-Ford algorithm in $O(mn)$ time, where m is the maximum number of edges at any time. As discussed before, in practice our bipartite graph is usually sparse with only $O(n)$ edges, which would bring the time complexity down to $O(n^2)$. Since every event means that one driver/request arrives, one request quits, or at least one driver/request leaves. Hence, there are $O(N)$ events in total.

Solving and maintaining P_2. First, because $\overline{U}_{rd}(\boldsymbol{\beta})$ is a linear function of the given arrival rate vector $\boldsymbol{\beta}$, this makes P_2 a linear program which can be solved efficiently. Moreover, the optimal solution of P_2 has a very simple greedy structure. For each request i, let k^i_d denote the slope of the variable λ^i_d in the objective of P_2. We have

$$k^i_d = \begin{cases} 0, & c_{r_i d} \geq c_{r_i d_j} \\ \frac{U'_{r_i d_j}(\boldsymbol{\lambda}) - c_{r_i d_j}}{\lambda_{<d_j}}, & c_{r_i d} < c_{r_i d_j}. \end{cases}$$

It is easy to see that in the optimal allocation, for each driver type d, we should allocate all of λ_d to the request i with the smallest slope k^i_d. This is a very simple greedy solution of P_2 which can be computed and maintained in $O(n)$ time.
To summarize, the time complexity of recomputing (solving and maintaining P_1 and P_2) is $O(n^2)$, and the time complexity of MULTI-REQ is $O(n^2 N)$.

Computing $\{\boldsymbol{T}^i\}$ and implementing Line 15 of MULTI-REQ. Finally, we explain how the matching event (i.e., Line 15 of MULTI-REQ) is detected and processed. Note that we do not need to check whether this condition is met for every request at every possible time t. Instead, each time when $\{\boldsymbol{T}^i\}$ needs to be updated, for each request i, we only need to compute one threshold $T^i_{d^i}$ using Theorem 1. This is the time when request i needs to be matched to its assigned driver. Then we check if the current time $t \geq T^i_{d^i}$ holds for any request i and proceed accordingly. If no matching happens at the moment, we only need to identify $t^* = \min_i T^i_{d^i}$ and register a future event that will be triggered at time t^*. In this way, the algorithm does not need to take any additional time to detect whether the condition in Line 15 is satisfied.

Experiments

In this section, we evaluate the performance of our algorithm using experiments on both synthetic and real-world datasets.[6]

Experiment Setup

We first construct a bipartite graph $G_0 = (\mathcal{R}, \mathcal{D}, E)$ that defines the arrival process and quitting process. We introduce two parameters: the *matching factor* C_L and the *waiting factor* T_L. C_L influences the value of edge cost c_{rd} and represents the relative importance of matching cost and waiting cost. Larger C_L means the matching cost carries more weight. T_L is only used by synthetic data and influences the quitting distribution of requests. Larger T_L means requests are more likely to wait for a longer time. The details of graph construction, arrival process and quitting process are deferred to the supplementary material.

We use a publicly available New York City taxi dataset (Donovan and Work 2014) (under CC0 license) and extract $100,000$ taxi trip records as our real-world data. The data processing details are also deferred to the supplementary material.

We first consider an important baseline—offline optimal (OPT) which denotes the optimal min-cost matching in the hindsight if we know the arrival and the quitting information of all requests and drivers. We recall the notation that we used before. Let R and D be the set of requests and drivers during the entire time horizon. For each request $i \in R$, let r_i be its type, t_i be its arrival time, and t^Q_i be its quitting time. For each driver $j \in D$, let d_j be its type and t_j its arrival time. This offline optimal solution can be computed by solving the following integer linear program. In this program, the binary variable x_{ij} indicates whether request i is matched to driver j. We also introduce a "null" driver j_0 and let $x_{ij_0} = 1$ indicate that request i quits. We set $D^+ = D \cup \{j_0\}$.

$$\min_{x_{ij}} \sum_{\substack{i \in R \\ j \in D}} (c_{r_i d_j} + \max(0, t_j - t_i)) x_{ij} + \sum_{i \in R} (c^Q_{r_i} + t^Q_i) x_{ij_0}$$

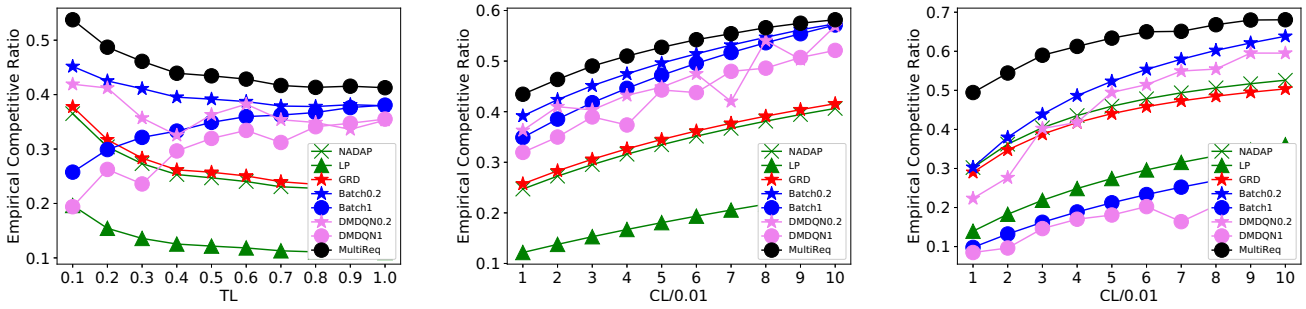
$$\text{s.t.} \sum_{j \in D^+} x_{ij} = 1 \qquad \forall i \in R$$

$$\sum_{i \in R} x_{ij} \leq 1 \qquad \forall j \in D$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in R, j \in D^+$$

Clearly this offline optimal solution is unattainable by any online algorithm and can only serve as a lower bound of the

[6] We use a computer with 2.2 GHz Intel Core i7 processor, 16 GB 1600 MHz DDR3 memory and Intel Iris Pro 1536 MB Graphics to run all the experiments. We use (Gurobi Optimization 2021) as our linear program solver in our code.



(a) Synthetic data $C_L = 0.01, T_L = 0.1, \ldots, 1.0$

(b) Synthetic data $T_L = 0.5, C_L = 0.01, \ldots, 0.1$

(c) Real-world data $C_L = 0.01, \ldots, 0.1$

Figure 1: Performance of different algorithms with synthetic and real-world data w.r.t. to T_L and C_L

total cost. Nevertheless, it could serve as a useful benchmark for our analysis and experiments.

Then we list the algorithms that are tested in experiments as below, including baseline algorithms and our algorithm.

- NADAP: This is the algorithm NADAP from (Dickerson et al. 2018b). This algorithm solves a two-sided online bipartite matching problem but without the waiting cost in their objective. We further modify the algorithm to allow requests to wait in the system if there is no available driver for it at its arrival (in (Dickerson et al. 2018b) such request is rejected right away).
- LP Matcher (LP): This is the LP-based algorithm 1 from (Collina et al. 2020). Their work assumes the quitting function follows a Poisson process, whereas in our experiment we use a step function to model the quitting function. Thus, when implementing LP, we set $\alpha_r = \frac{1}{Q_r}$ as the departure rate of request type r and $\alpha_d = 0$ as the departure rate of driver type d.
- Greedy (GRD): When a request/driver arrives, immediately match it to the nearest available driver/request.
- Batch-w: This is an algorithm that divides the time horizon into small length w windows. The unit of w is minutes. At the end of each window, a min-cost bipartite matching is applied to all available requests and drivers. We test with $w = 0.2$ and $w = 1$.
- DMDQN-w: The algorithm Delayed-M-DQN from (Ke et al. 2020). It is based on Batch-w (w has the same meaning in DMDQN and Batch). The difference is that the min-cost bipartite matching at the end of each window is applied to all available drivers and requests in matching pool which is decided by deep reinforcement learning.
- MULTI-REQ: Refer to our algorithm from Section .

We conduct our experiment on 100 test cases with both synthetic data and real-world data. For each test case, we sample an arrival sequence s with 1000 arrivals (including both requests and drivers). Each arrival contains the arrival time, arrive type, and quitting time if it is a request. For synthetic data, the arrival time and type are generated according to the arrival process. For real-world data, we choose 1000 consecutive arrivals from the dataset as the arrival sequence.

Let $\mathsf{OPT}(s), \mathsf{ALG}(s)$ denote the cost of a given algorithm with input sequence s and $\mathsf{ALG} \in \{$NADAP, LP, GRD, Batch-0.2, Batch-1, DMDQN-0.2, DMDQN-1, MULTI-REQ $\}$. We use *empirical competitive ratio* (ECR) to measure the performance of an algorithm.

$$\text{ECR} = \frac{\sum_{s \in S} \mathsf{OPT}(s)}{\sum_{s \in S} \mathsf{ALG}(s)}, \tag{1}$$

where S denotes the set of all sampled sequences. We use ECR as our performance measure because when S is the set of all possible sequences, Equation 1 becomes the *competitive ratio*, which is a standard evaluation metric for online algorithms.

Results

ECR Evaluation. Figure 1 demonstrates the performance (ECR) of all baseline algorithms with the synthetic dataset and real-world dataset. As one can see from the figures, MULTI-REQ outperforms the baseline algorithms NADAP, LP and GRD consistently by 10% - 20%. This shows that identifying the right waiting time of a request can greatly help to improve the overall efficiency of the driver-request assignment. Compared to Batch-0.2 and Batch-1, our algorithm outperforms Batch-0.2 and Batch-1 especially when C_L and T_L are small. We can also notice the performance of Batch-w is always better than DMDQN-w with same window size w.

Figure 1(a) shows the algorithm performances when T_L varies from 0.1 to 1.0 (and C_L is set to 0.01). When T_L becomes larger, it means there are more possible new drivers arriving during the request's waiting window. Consequently, algorithms that do not consider waiting time (NADAP, LP and GRD) or only wait for short time (Batch-0.2) exhibit a decrease in their performances. For Batch-1, the large window size of 1 minutes means there are requests arriving and quitting in the window, and these requests will not be matched by the algorithm. Note that we also see a decrease in the performance of our algorithm MULTI-REQ. This is because when T_L becomes larger, the offline optimal algorithm

Outcomes	OPT	NADAP	LP	GRD	Batch-0.2	Batch-1	DMDQN-0.2	MULTI-REQ
No Wait	430.55	437.20	186.55	455.25	0	0	0	346.80
Wait&Match	53.25	29.70	273.05	7.40	479.2	462.85	471.1	131.05
Wait&Quit	16.30	33.20	40.05	37.45	20.90	37.25	29	22.25

Table 1: Average number of requests with each outcome.

will have a bigger advantage by knowing all future events beforehand. Nevertheless, our algorithm maintains a strong ECR and can still outperform all the baseline algorithms.

Figure 1(b) and 1(c) record the algorithm performances when C_L varies from 0.01 to 0.1 for synthetic data (T_L is set to 0.5) and real-world data. A larger value of C_L means the driver pick-up distances would matter more in the overall assignment costs. Because all baseline algorithms put a higher priority in minimizing the matching cost, their relative performances all increase with C_L. With that being said, our algorithm is able to hold a consistent edge over the baseline algorithms.

Request Outcomes. We also provide statistics on the different request outcomes in the output of each algorithm. In our model, each request has three possible ending outcomes: (1) be assigned to a driver immediately upon its arrival, (2) wait for some time then be assigned to a driver, and (3) leave the system unassigned. Table 1 shows the average number of requests with each ending outcome in a single test case in each algorithm (with real-world dataset and parameters set as $C_L = 0.01$).

From the table, one can see that GRD and NADAP both try to assign requests to drivers as soon as possible. As a result, in their solutions the number of requests assigned without waiting is larger than that in other algorithms. On the other hand, LP does not contain the waiting cost component in its objective function, therefore its solution will naturally favor assignments with small matching costs and large waiting costs. Because Batch-w and DMDQN-w only make matches at the end of each window, essentially all requests need to wait. Compared to these algorithms, our algorithm MULTI-REQ is able to find the right amount of waiting time for each request, therefore achieving a more balanced assignment with better overall cost. Our output also have the second least number of unmatched requests compared to other baselines.

Conclusion

In this paper, we put forward a novel driver-request assignment model for on-demand ridesourcing. Our model allows real-time decision-making and takes account of requests voluntarily leaving the platform, therefore providing a more realistic view of ridesourcing assignments in practice. We then propose an efficient assignment algorithm that could balance requests' waiting times and the driver's pick-up distances, and we demonstrate its effectiveness on both synthetic and real-world datasets.

One interesting future working direction is to extend this model with relaxed assumptions. For example, in future works we can also allow drivers to leave the platform voluntarily as well. Another extension is to introduce ride-sharing (i.e., multiple requests sharing the same vehicle) to the model and study the dynamic ride-sharing assignment problem.

Acknowledgments

This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-045).

References

Aggarwal, G.; Goel, G.; Karande, C.; and Mehta, A. 2011. Online vertex-weighted bipartite matching and single-bid budgeted allocations. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, 1253–1264. SIAM.

Alonso-Mora, J.; Samaranayake, S.; Wallar, A.; Frazzoli, E.; and Rus, D. 2017. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proceedings of the National Academy of Sciences*, 114(3): 462–467.

Anshelevich, E.; Chhabra, M.; Das, S.; and Gerrior, M. 2013. On the social welfare of mechanisms for repeated batch matching. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27.

Aouad, A.; and Saritaç, Ö. 2020. Dynamic stochastic matching under limited time. In *The 21st ACM Conference on Economics and Computation*, 789–790.

Ashlagi, I.; Azar, Y.; Charikar, M.; Chiplunkar, A.; Geri, O.; Kaplan, H.; Makhijani, R.; Wang, Y.; and Wattenhofer, R. 2017. Min-cost bipartite perfect matching with delays. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2017)*, 81: 1–1.

Azar, Y.; Chiplunkar, A.; and Kaplan, H. 2017. Polylogarithmic bounds on the competitiveness of min-cost perfect matching with delays. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1051–1061. SIAM.

Azar, Y.; and Fanani, A. J. 2020. Deterministic min-cost matching with delays. *Theory of Computing Systems*, 1–21.

Banerjee, S.; Johari, R.; and Riquelme, C. 2015. Pricing in ride-sharing platforms: A queueing-theoretic approach. In *Proceedings of the Sixteenth ACM Conference on Economics and Computation*, 639–639.

Bei, X.; and Zhang, S. 2018. Algorithms for trip-vehicle assignment in ride-sharing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1.

Bertsimas, D.; Jaillet, P.; and Martin, S. 2019. Online vehicle routing: The edge of optimization in large-scale applications. *Operations Research*, 67(1): 143–162.

Caramia, M.; Italiano, G. F.; Oriolo, G.; Pacifici, A.; and Perugia, A. 2001. Routing a fleet of vehicles for dynamic combined pick-up and deliveries services. In *Operations Research Proceedings*, 3–8. Springer.

Collina, N.; Immorlica, N.; Leyton-Brown, K.; Lucier, B.; and Newman, N. 2020. Dynamic Weighted Matching with Heterogeneous Arrival and Departure Rates. In *International Conference on Web and Internet Economics*, 17–30. Springer.

Colorni, A.; and Righini, G. 2001. Modeling and optimizing dynamic dial-a-ride problems. *International transactions in operational research*, 8(2): 155–166.

Cordeau, J.-F.; and Laporte, G. 2007. The dial-a-ride problem: models and algorithms. *Annals of operations research*, 153(1): 29–46.

Coslovich, L.; Pesenti, R.; and Ukovich, W. 2006. A two-phase insertion technique of unexpected customers for a dynamic dial-a-ride problem. *European Journal of Operational Research*, 175(3): 1605–1615.

Dickerson, J.; Sankararaman, K.; Srinivasan, A.; and Xu, P. 2018a. Allocation problems in ride-sharing platforms: Online matching with offline reusable resources. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1.

Dickerson, J. P.; Sankararaman, K. A.; Srinivasan, A.; and Xu, P. 2018b. Assigning tasks to workers based on historical data: Online task assignment with two-sided arrivals. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

Donovan, B.; and Work, D. 2014. New York City Taxi Data (2010-2013). *Dataset, http://dx. doi. org/10.13012/J8PN93H8*.

Emek, Y.; Kutten, S.; and Wattenhofer, R. 2016. Online matching: haste makes waste! In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, 333–344.

Feng, J.; Gluzman, M.; and Dai, J. G. 2021. Scalable deep reinforcement learning for ride-hailing. In *2021 American Control Conference (ACC)*, 3743–3748. IEEE.

Gurobi Optimization, L. 2021. Gurobi Optimizer Reference Manual.

Huang, Z.; Kang, N.; Tang, Z. G.; Wu, X.; Zhang, Y.; and Zhu, X. 2018. How to match when all vertices arrive online. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of computing*, 17–29.

Huang, Z.; Peng, B.; Tang, Z. G.; Tao, R.; Wu, X.; and Zhang, Y. 2019. Tight competitive ratios of classic matching algorithms in the fully online model. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2875–2886. SIAM.

Karp, R. M.; Vazirani, U. V.; and Vazirani, V. V. 1990. An optimal algorithm for on-line bipartite matching. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 352–358.

Ke, J.; Xiao, F.; Yang, H.; and Ye, J. 2020. Learning to delay in ride-sourcing systems: a multi-agent deep reinforcement learning framework. *IEEE Transactions on Knowledge and Data Engineering*.

Lee, D.-H.; Wang, H.; Cheu, R. L.; and Teo, S. H. 2004. Taxi dispatch system based on current demands and real-time traffic conditions. *Transportation Research Record*, 1882(1): 193–200.

Lesmana, N. S.; Zhang, X.; and Bei, X. 2019. Balancing efficiency and fairness in on-demand ridesourcing. *Advances in Neural Information Processing Systems*, 32.

Lowalekar, M.; Varakantham, P.; and Jaillet, P. 2021. Zone pAth Construction (ZAC) based Approaches for Effective Real-Time Ridesharing. *Journal of Artificial Intelligence Research*, 70: 119–167.

Ma, S.; Zheng, Y.; and Wolfson, O. 2013. T-share: A large-scale dynamic taxi ridesharing service. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 410–421. IEEE.

Mehta, A.; Saberi, A.; Vazirani, U.; and Vazirani, V. 2007. Adwords and generalized online matching. *Journal of the ACM (JACM)*, 54(5): 22–es.

Miao, F.; Han, S.; Lin, S.; Stankovic, J. A.; Zhang, D.; Munir, S.; Huang, H.; He, T.; and Pappas, G. J. 2016. Taxi dispatch with real-time sensing data in metropolitan areas: A receding horizon control approach. *IEEE Transactions on Automation Science and Engineering*, 13(2): 463–478.

Nanda, V.; Xu, P.; Sankararaman, K. A.; Dickerson, J.; and Srinivasan, A. 2020. Balancing the tradeoff between profit and fairness in rideshare platforms during high-demand hours. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 2210–2217.

Xu, Z.; Li, Z.; Guan, Q.; Zhang, D.; Li, Q.; Nan, J.; Liu, C.; Bian, W.; and Ye, J. 2018. Large-scale order dispatch in on-demand ride-hailing platforms: A learning and planning approach. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 905–913.

Zhang, R.; and Pavone, M. 2016. Control of robotic mobility-on-demand systems: a queueing-theoretical perspective. *The International Journal of Robotics Research*, 35(1-3): 186–203.

Zukerman, M. 2013. Introduction to queueing theory and stochastic teletraffic models. *arXiv preprint arXiv:1307.2968*.