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 (Colomn 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 (Bej 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; Dickinson 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 Saqiq (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 that 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, Chipfunkar, 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 $G_0 = (R, D, E)$ that is known to the algorithm. Here $R = \{r | 1 \leq r \leq N\}$ and $D = \{d | 1 \leq d \leq M\}$ are the type spaces of requests and drivers, respectively, with $|R| = N$ and $|D| = M$. $E$ 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 \( c_{r,d}\). 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 \( d \in D \), 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 \( \lambda_d \). This is a standard assumption for driver arrivals that has been considered in many works (Collina et al. 2020; Aouad and Saritac 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 arrival 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 \( F(t) \) denotes 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 driver 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 \( c_{r,d} \) 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 \( c_i^Q \). We assume \( c_i^Q > c_{r,d} \) 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 \( \lambda_i \), and we denote the vector \( \Lambda = (\lambda_1, \lambda_2, \ldots, \lambda_M) \).

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\(_m\) \((\Lambda, T = (T_1, T_2, \ldots, T_{M+1}) \) 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 \( c_{(M+1)} = \infty \) 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

\textsc{Single-Req}(\lambda, T)

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

1: \(m \leftarrow M + 1\) // \(m\) represents the best available driver type so far
2: \textbf{while} time \(t\) grows continuously starting from 0 \textbf{do}
3: \textbf{if} A type \(d_i\) driver arrives with \(i < m\) \textbf{then}
4: \(m \leftarrow i\)
5: \textbf{end if}
6: \textbf{if} \(t \geq T_m\) \textbf{then}
7: match request \(r\) to the \(d_m\) driver
8: \textbf{return}
9: \textbf{end if}
10: \textbf{end while}

Finding the optimal threshold times \(T^*_j\). Next we provide a characterization of the optimal thresholds in algorithm \textsc{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 \textsc{Single-Req}(\(\lambda, T = (T_1^*, \ldots, T_{M+1}^*)\)), where \(T_j^*\) is the smallest nonnegative value satisfies
\[q(T_j^*) \geq \sum_{i=1}^{j-1} \lambda_i (b_j - b_i) - 1 \quad c_q - b_j,\]
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.

\textbf{Proof.} Consider the scenario in which \(j\) is currently the best available driver type, and we are at time \(T_j^*\). 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 \(\Delta\), 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 \(\Delta\) approach 0, we have that \(T_j^*\) is the smallest value that satisfies
\[q(T_j^*) \geq \sum_{i=1}^{j-1} \lambda_i (b_j - b_i) - 1 \quad c_q - b_j.\]

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. \(q(t)(c_q - b_j)\) 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. \(\lambda_j (b_j - b_i)\) is the probability density of the arriving distribution of type \(d_j\) driver. Therefore, \(\lambda_j (b_j - b_i)\) 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 \(T_j^*\).

\textbf{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 \textsc{Single-Req}, and the threshold time \(T_j^*\) is the smallest nonnegative value satisfying
\[q(T_j^*) \geq \sum_{i=1}^{j-1} \lambda_i (b_j - b_i) - h'(t) \quad c_q - b_j,\]
where \(h'(t)\) is the derivative of \(h(t)\).

\textbf{Multiple Requests.} In this section we investigate the general case with multiple heterogeneous requests and drivers. Following the single-request case in Section \(\ref{sec:algorithm}\), 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 $r$ 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 $\lambda$ into $n$ independent Poisson subprocesses with rates $\lambda_i(1 \leq i \leq n)$ as long as it satisfies $\sum \lambda_i = \lambda$. 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($\lambda$)**

**Input:** $M$ driver types with arrival rates $\lambda = (\lambda_1, \ldots, \lambda_M)$; time $t$ grows continuously starting from 0, do
1: **while** time $t$ grows continuously starting from 0 do
2: $R \leftarrow$ currently available requests
3: $D \leftarrow$ currently available drivers
4: if a new request $i_{\text{new}}$ arrives then
5: create a new SINGLE-REQ$_{\text{new}}$ starting from $t$
6: end if
7: if a request $i \in R$ quits then
8: terminate SINGLE-REQ$_i$
9: end if
10: if any arrival or quit event happens then
11: recompute $\{X_i\}$ and $\{d^i\}$, $\forall i \in R$
12: recompute thresholds $T^i$ from Thm 1, $\forall i \in R$
13: update $\{X_i\}, \{d^i\}$, and $\{T^i\}$ in SINGLE-REQ$_i(\lambda^i, T^i, d^i), \forall i \in R$
14: end if
15: **while** there exists request $i$ with $t \geq T^i_d$ in SINGLE-REQ$_i$ do
16: pick $i$ with the smallest $T^i_d$, among all such requests
17: match request $i$ to the corresponding driver $d^i$
18: terminate SINGLE-REQ$_i$
19: recompute and update $\{X_i\}$ and $\{T^i\}, \forall i \in R$
20: end while
21: end while

Given this algorithm framework, we have two important tasks left: how to allocate the available drivers in $D$ and the driver arrival rates $\lambda$ 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 $D^+ = D \cup \{j_0\}$.
- $U^\beta_{rd}(t)(r \in R, d \in D)$: 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 $\beta$.

Let $P_0$ denote the optimization problem. Unfortunately, we cannot solve $P_0$ directly because $U^\lambda_{ri}(t)$ can be very

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0 : \min_{x_{ij}, \lambda^i} \sum_{i \in R} \sum_{j \in D^+} x_{ij} U^\lambda_{ri,j}(t - t_i)$</td>
<td>$P_1 : \min_{x_{ij}} \sum_{i \in R} \sum_{j \in D^+} x_{ij} \bar{U}_{r,j}$</td>
<td>$P_2 : \min_{\lambda^i} \sum_{i \in R} \bar{U}_{r,i}^\lambda(t)$</td>
</tr>
<tr>
<td>s.t.</td>
<td>s.t.</td>
<td>s.t.</td>
</tr>
<tr>
<td>$\sum_{j \in D^+} x_{ij} = 1 \quad \forall i \in R$</td>
<td>$\sum_{i \in R} x_{ij} = 1 \quad \forall i \in R$</td>
<td>$\sum_{\lambda^i} \lambda^i = \lambda$</td>
</tr>
<tr>
<td>$\sum_{i \in R} x_{ij} \leq 1 \quad \forall j \in D$</td>
<td>$\sum_{i \in R} x_{ij} \leq 1 \quad \forall j \in D$</td>
<td>$\forall i \in R, j \in D^+$</td>
</tr>
<tr>
<td>$x_{ij} \in {0, 1} \quad \forall i \in R, j \in D^+$</td>
<td>$x_{ij} \in {0, 1} \quad \forall i \in R, j \in D^+$</td>
<td></td>
</tr>
<tr>
<td>$\lambda^i \geq 0 \quad \forall i \in R$</td>
<td>$\lambda^i \geq 0 \quad \forall i \in R$</td>
<td></td>
</tr>
</tbody>
</table>

$\lambda_i$ can be done by assigning each arrival to subprocess $i$ with probability $\lambda_i / \lambda$.

We make two adjustments to the single-request algorithm SINGLE-REQ$_i(\lambda^i, T^i)$ from Section. First, we allow the single
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 $\bar{U}_{rd}$, which will be discussed later in details, as an approximation of all $U_{rd}^\beta(t)$ with different $\beta$ and $t$, and use these values to compute the best request-driver matching $x^*$. We denote this first-stage optimization problem as $P_1$.

Next, given the matching $x^*$, for each $i \in R$, there is only one driver $d_i$ with $x_{rd_i} = 1$. We then use another linear approximation $\bar{U}_{rd}^\beta(\beta)$, which also will be discussed later, to approximate $U_{rd}^\beta(t)$ (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 $\bar{U}_{rd}$ in $P_1$ and $U_{rd}(\beta)$ in $P_2$. We want our design to satisfy two properties: (1) they are reasonable approximations of the actual cost function $U_{rd}^\beta(t)$, and (2) they can be computed and maintained efficiently.

We start with $U_{rd}(\beta)$. Fix any request type $r$ and driver type $d$. Note that $U_{rd}^\beta(t)$ 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 $T_{rd}^\beta(\beta)$. However, the expected cost of the optimal algorithm does not have a simple closed-form and may be affected by the value of $\beta$ 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 $T_{rd}^\beta(\beta)$ and quit right after. Third, we also assume a simple strategy: whenever a driver better than $d$ arrives before time $T_{rd}^\beta(\beta)$, match it to the request immediately. If there are no such drivers, match the request to the only available driver at time $T_{rd}^\beta(\beta)$. Let $U_{rd}(\beta)$ be the expected cost function with all these simplifications. We will find the explicit solution of $U_{rd}(\beta)$, and then approximate it with a linear function $\bar{U}_{rd}(\beta)$ with arrival rates $\beta$ as the function parameter.

We denote

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

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 $\beta_{d}$, 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

$$C(x) = \begin{cases} \beta_{<d} + x, & x \leq T_{rd}^\beta(\beta) \\ \beta_{rd} + T_{rd}^\beta(\beta), & x > T_{rd}^\beta(\beta) \end{cases}$$

From here we can then derive the expected cost function

$$U_{rd}(\beta) = \int_0^\infty \beta_{<d} \exp(-\beta_{<d} x) C(x)dx$$

$$= (\beta_{rd} - \beta_{<d}) \exp(-\beta_{<d} T_{rd}^\beta(\beta)) + \beta_{<d}$$

where $\beta_{<d} = c_{<d} + \frac{1}{\beta_{<d}}$.

Having obtained $U_{rd}(\beta)$, our final task is to approximate it with a linear function. It is easy to see that $U_{rd}(\beta) = c_{rd}$.

We will use the linear function $\bar{U}_{rd}(\beta)$ that goes through $(0, c_{rd})$ and $(\lambda, U_{rd}(\lambda))$ to approximate $U_{rd}(\beta)$.

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

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

**Approximations in $P_1$ and $P_2$**

- In $P_1$, we use:
  $$\bar{U}_{rd} = \frac{1}{2}(U_{rd}(0) + U_{rd}(\lambda)) = \frac{1}{2}(c_{rd} + U_{rd}(\lambda)).$$

- In $P_2$, we use:
  $$\bar{U}_{rd}(\beta) = U_{rd}(\lambda) - c_{rd} \frac{\beta_{<d}}{\lambda_{<d}} c_{rd} + c_{rd}.$$

**Implementation**

We now discuss the implementation of the multiple-request algorithm MULTI-REQ. Although the algorithm describes a continuous time process, during which $\{\lambda_i\}, \{d_i\}$, and $\{T_i\}$ 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 $\{\lambda_i\}$ 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 $n$ 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 whose optimal solution
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 \( T_{ij}^d \), in the subproblem SINGLE-REQ, for some request \( i \): this means we need to assign request \( i \) to its matching driver \( d \) in the optimal matching at this time. Both request \( i \) and driver \( d \) 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 \( U_{r,d}(\beta) \) is a linear function of the given arrival rate vector \( \beta \), 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 \( \lambda^i_d \) in the objective of \( P_2 \). We have

\[
\begin{align*}
    k^i_d = \begin{cases} 
        0, & \text{if } D_t \geq \lambda^i_d, \\
        U_{r,d}(\lambda) - c_{r_dij} / c_{r_dij}, & \text{otherwise} 
    \end{cases}
\end{align*}
\]

It is easy to see that in the optimal allocation, for each driver type \( d \), we should allocate all of \( \lambda^i_d \) 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 \( O(n^2N) \).

**Computing \( \{T^i\} \) 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 \( \{T^i\} \) needs to be updated, for each request \( i \), we only need to compute one threshold \( T_{d_i}^i \) 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 \( t \geq T_{d_i}^i \) holds for any request \( i \) and proceed accordingly. If no matching happens at the moment, we only need to identify \( t^* = \min_{d_i} T_{d_i}^i \) 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 \( G_0 = (R, D, E) \) 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 CCO 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 \( t^Q_i \) 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 \( D^+ = D \cup \{j_0\} \).

\[
\begin{align*}
    \min_{x_{ij}} & \sum_{i \in R} (c_{r_i d_j} + \max(0, t_j - t_i))x_{ij} + \sum_{i \in R} (c_{r_i j} + t^Q_i) x_{ij_0} \\
    \text{s.t.} & \sum_{j \in D^+} x_{ij} = 1 \quad \forall i \in R \\
    & \sum_{i \in R} x_{ij} \leq 1 \quad \forall j \in D \\
    & x_{ij} \in \{0, 1\} \quad \forall i \in R, j \in D^+ 
\end{align*}
\]

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.
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 2.

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

\[
\text{ECR} = \frac{\sum_{s \in S} \text{OPT}(s)}{\sum_{s \in S} \text{ALG}(s)},
\]

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

![Figure 1: Performance of different algorithms with synthetic and real-world data w.r.t. to \( T_L \) and \( C_L \)](image-url)
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 $C_L = 0.01$).

<table>
<thead>
<tr>
<th>Outcomes</th>
<th>OPT</th>
<th>NADAP</th>
<th>LP</th>
<th>GRD</th>
<th>Batch-0.2</th>
<th>Batch-1</th>
<th>DMDQN-0.2</th>
<th>MULTI-REQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Wait</td>
<td>430.55</td>
<td>437.20</td>
<td>186.55</td>
<td>455.25</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>346.80</td>
</tr>
<tr>
<td>Wait&amp;Match</td>
<td>53.25</td>
<td>29.70</td>
<td>273.05</td>
<td>7.40</td>
<td>479.2</td>
<td>462.85</td>
<td>471.1</td>
<td>131.05</td>
</tr>
<tr>
<td>Wait&amp;Quit</td>
<td>16.30</td>
<td>33.20</td>
<td>40.05</td>
<td>37.45</td>
<td>20.90</td>
<td>37.25</td>
<td>29</td>
<td>22.25</td>
</tr>
</tbody>
</table>

Table 1: Average number of requests with each outcome.

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


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.


