Wait to be Faster: a Smart Pooling Framework for Dynamic Ridesharing

Xiaoyao Zhong *, Jiabao Jin *, Peng Cheng *, Wangze Ni ◇, Libin Zheng †, Lei Chen ◇, Xuemin Lin ‡
*East China Normal University, Shanghai, China
◇HKUST(GZ) and HKUST, Guangzhou and Hong Kong SAR, China
†Sun Yat-sen University, Guangzhou, China
‡Shanghai Jiaotong University, Shanghai, China
{xiaoyao.zhong, jiabaojin}@stu.ecnu.edu.cn; pcheng@sei.ecnu.edu.cn; wniab@cse.ust.hk;
zhenglb6@mail.sysu.edu.cn; leichen@cse.ust.hk; xuemin.lin@gmail.com

***Abstract*—Ridesharing services, such as Uber or Didi, have attracted considerable attention in recent years due to their positive impact on environmental protection and the economy. Existing studies require quick responses to orders, which lack the flexibility to accommodate longer wait times for better grouping opportunities. In this paper, we address a NP-hard ridesharing problem, called *Minimal Extra Time RideSharing* (METRS), which balances waiting time and group quality (i.e. detour time) to improve riders' satisfaction. To tackle this problem, we propose a novel approach called WATTER (WAit To be fasTER), which leverages an order pooling management algorithm allowing orders to wait until they can be matched with suitable groups. The key challenge is to customize the extra time threshold for each order by reducing the original optimization objective into a convex function of threshold, thus offering a theoretical guarantee to be optimized efficiently. We model the dispatch process using a Markov Decision Process (MDP) with a carefully designed value function to learn the threshold. Through extensive experiments on three real datasets, we demonstrate the efficiency and effectiveness of our proposed approaches.**

I. Introduction

With the increasing popularity of the sharing economy, more and more ridesharing platforms are emerging to facilitate people's lives, such as Uber and Didi. The ridesharing service is to group riders with overlapping travel routes and similar time schedules, and then assign them to workers to serve. It not only lowers prices for riders and saves fuel consumption for workers, but also eases traffic congestion and reduces carbon dioxide emissions.

In ridesharing, the platform may have different optimization goals: (1) maximizing platform revenue [1], [2], [3]; (2) minimizing the total travel distance of workers [4], [5], [6]; (3) maximizing the number of served orders [7], [2]. To achieve a high service rate, orders will be dispatched even if they result in worse satisfaction, and will only be rejected when they cannot be served in the extreme cases.

In dynamic ridesharing, existing studies propose two processing modes: online-based mode and batch-based mode. Online-based methods [8], [9], [10] provide a real-time response to each order. Batch-based methods [11], [12], [2] usually group the orders within a batch (i.e., a time window of 5 seconds) based on specific combination strategy and assign the groups to workers. We observe from real datasets that

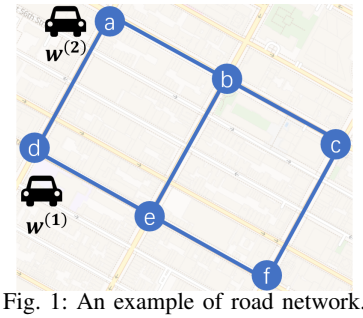


Fig. 1: An example of road network.

TABLE I: Online arriving orders.

Order $o^{(i)}$	Release Time (s) $t^{(i)}$	Batch round $t^{(i)}/10$	Pick-up Location $l_p^{(i)}$	Drop-off Location $l_d^{(i)}$
$o^{(1)}$	5	0	a	c
$o^{(2)}$	8	0	d	e
$o^{(3)}$	10	0	c	a
$o^{(4)}$	12	1	d	f
$o^{(5)}$	17	1	b	c

orders can wait for a while (e.g., 10 seconds) to get a better grouping result with less travel costs. We illustrate this with the following example:

Example 1. We assume that there are two idle workers and that orders $o^{(1)} \sim o^{(5)}$ arrive at the platform in ascending order of $t^{(i)}$. The optimization objective is to minimize the total travel time of the workers. The road network consists of six nodes and seven edges. Each edge represents a road travel time of 1 minute. The information of orders is shown in the Table I. For the batch-based method, let's consider a case where the batch size is in the 10 seconds. For instance, orders $o^{(1)} \sim o^{(3)}$ are in batch round 0, and orders $o^{(4)} \sim o^{(5)}$ are in another batch round 1. As a result, $o^{(1)}$ and $o^{(2)}$ will be grouped together, and $o^{(4)}$ and $o^{(5)}$ will be grouped together, hence resulting in a total travel time of $4+4=8$ mins. As for the online-based method, the platform will group $o^{(1)}$ and $o^{(2)}$ together, and group $o^{(3)}$ and $o^{(4)}$ together, resulting in a total travel time of $4+5=9$ mins. However, the best match for $o^{(1)}$ is actually $o^{(5)}$, and the best match for $o^{(2)}$ is $o^{(4)}$. This results in a total travel time of $2+2=4$ mins. Compared to the previous two frameworks, this pooling-then-grouping strategy only causes the orders to wait slightly longer but greatly reduces the total travel time.

Challenge: Unlike existing studies that only respond orders immediately or in a static mini-batch time, *can we allow orders to wait for a better grouping opportunity results in shorter total travel times?* Intuitively, the longer an order waits and the more other orders arrive, the higher probability of it being grouped with more suitable orders, which leads to that its total travel cost/time can be reduced. As it is difficult to directly predict the arriving orders in the next several seconds, the main challenge is to determine the optimal waiting/pooling time before dispatching the order.

To address the challenge, we formalize a new problem Minimal Extra Time Ride Sharing (METRS), which takes the waiting times and detour times into consideration.

We propose a novel framework called WATTER (WAit To be fasTER), which leverages a *order pooling management algorithm* to maintain the orders and the shareability relationships in the temporal shareability graph. We propose an effective average extra time threshold-based grouping strategy that assigns a threshold of *expected extra time* to each order. We theoretically prove that the optimization objective of METRS problem can be reduced to a convex function of extra time threshold, providing a theoretical guarantee for optimization effectiveness. We take spatio-temporal environment into consideration, then model the decision-making process of holding or dispatching orders in the pool as a Markov Decision Process (MDP). Historical data is used to offline generate training experience by simulating the dispatch process of the framework incorporated with the proposed grouping strategy. We utilize this experience to train the value function in MDP, which is then used as an estimation of the expected extra time in the online decision-making process. To summarize, we make the following contributions in the paper:

- We formulate the METRS problem to balance waiting response time and detour time and prove its hardness in Section II.
- We introduce the order pooling management algorithm and related algorithms in Section III.
- We devise a average extra time threshold-based grouping strategy with theoretical analysis in Section V.
- We model the dispatch process and establish a offline reinforcement learning model combined with the online threshold-based strategy to make decisions in Section VI.
- Extensive experiments on real datasets is conducted in Section VII.

II. Problem Definition

A. Preliminaries

Definition 1. (Order) An order is denoted by $o^{(i)} = \langle l_p^{(i)}, l_d^{(i)}, c^{(i)}, t^{(i)}, \tau^{(i)}, \eta^{(i)} \rangle$. The order $o^{(i)}$ is released at timestamp $t^{(i)}$ and contains $c^{(i)}$ riders. The order asks to deliver riders from the pick-up location $l_p^{(i)}$ to the drop-off location $l_d^{(i)}$ before the deadline $\tau^{(i)}$. The platform needs to give response to order within waiting time limit $\eta^{(i)}$.

An *order group* is a collection of orders that can be represented as $g = \{o^{(1)}, o^{(2)}, ..., o^{(|g|)}\}$, where $|g|$ denotes the number of orders in the group. Note that the waiting time limit $\eta^{(i)}$ is a customized parameter just to indicate the preferred limit waiting time of $o^{(i)}$, which is not a constraint. In our paper, if $o^{(i)}$ has waited more than $\eta^{(i)}$ time, it should be dispatched as soon as possible. Only when $o^{(i)}$ cannot be delivered to its destination before $\tau^{(i)}$, it is considered rejected.

Definition 2. (Worker) A worker can be denoted by $w^{(j)} = \langle l^{(j)}, k^{(j)}, a^{(j)} \rangle$, where $l^{(j)}$ is the worker's current location, $k^{(j)}$ is the vehicle capacity, and $a^{(j)}$ is worker's availability.

The availability $a^{(j)}$ can either be *idle* (e.g., waiting for an assignment) or *busy* (e.g., delivering an order group). In this paper, we assume that a worker can only deliver one order group at a time.

Let $G^{(j)}$ denote the set of order groups that worker $w^{(j)}$ served in a day. Then, the set $G = \cup_{w^{(j)} \in W} G^{(j)}$ is composed of all served order groups. Given the set O of all orders, let $O^+ = \cup_{g \in G} \cup_{o^{(i)} \in g} o^{(i)} \subseteq O$ be the successfully served orders in G. Let $O^- = O - O^+$ denote the set of all rejected orders.

Definition 3. (Route) The route is an ordered sequence of locations denoted by $L = \langle l_1, l_2, ..., l_{|L|} \rangle$, where each l_i represents a location on the road network.

An order group g can generate a route L by aligning a sequence that includes the pick-up and drop-off locations of all orders in g. Then, the assigned worker $w^{(j)}$ travels to location l_1 and follows the route to serve the orders in the group. We use $L^{(i)}$ to denote the sub-route starting from l_1, passing through $l_p^{(i)}$, and ending at $l_d^{(i)}$. The travel cost of the route can be calculated as $T(L) = \sum_{k=1}^{|L|-1} cost(l_k, l_{k+1})$, where $cost$ is the shortest travel time of two locations.

Definition 4. (Response Time) For a given order $o^{(i)}$, let $t_n^{(i)}$ denote the time when the platform notifies the grouping result, then the response time of $o^{(i)}$ is $t_r^{(i)} = t_n^{(i)} - t^{(i)}$, which refers to the waiting time from the order being released to being notified with assignment.

In practice, riders have limited patience for waiting. Long response times may cause riders to cancel their orders, resulting in potential revenue loss for the platform. However, minimizing response time alone may lead to missing the potential future properer riders and results in long detours, which also can sacrifice the satisfactions of riders and drive them to choose other transportation methods or platforms. Thus, it is important to smartly balance the response times and detours without clearly knowing future orders.

Definition 5. (Detour Time) For a given order $o^{(i)}$ in order group g, let L be the generated route for g, the detour time $t_d^{(i)}$ of $o^{(i)}$ is denoted by $t_d^{(i)} = T(L^{(i)}) - cost(l_p^{(i)}, l_d^{(i)})$, which refers to the additional time cost incurred by sharing the route with other riders compared to the minimal shortest time $cost(l_p^{(i)}, l_d^{(i)})$.

Both response time and detour time can be considered as the extra cost of riders in taking the ridesharing service, which is the major factor to affect the riders' satisfaction. In this

paper, we define extra time as a unified metric to reflect riders' satisfaction.

Definition 6. (Extra Time) The extra time $t_e^{(i)}$ of order $o^{(i)} \in g$ is defined as:

$$t_e^{(i)} = \alpha t_d^{(i)} + \beta t_r^{(i)} \tag{1}$$

where α and β are coefficients used for trade-off between the detour time $t_d^{(i)}$ and the response time $t_r^{(i)}$.

We offer the flexibility to adjust the weight in definition. By setting $\alpha = 1$ and $\beta = 1$, $t_e^{(i)}$ represent the real extra time of the rider, compared to the shortest travel time of the order.

B. Problem Definition

We defined the Minimal Extra Time RideSharing problem:

Definition 7. (METRS Problem) Given an online order set O and a worker set W, the METRS problem is to find a set of shareable order groups G for each order $o^{(i)} \in O$ and assign each group of orders with a suitable workers, such that total extra time of orders in the platform $\Phi(W, O)$ is minimized:

$$\min_G \Phi(W, O) = \sum_{o^{(i)} \in O^+} t_e^{(i)} + \sum_{o^{(j)} \in O^-} p^{(j)} \tag{2}$$

where $p^{(j)}$ indicates the penalty of $o^{(j)}$ if it is rejected. An order group g is *shareable* if and only if it can generate a *feasible* route L with a available worker $w^{(j)}$ satisfying the following constraints:

1) Sequential constraint: $\forall o^{(i)} \in g$, it has $l_p^{(i)} = l_x \in L$ and $l_d^{(i)} = l_y \in L$, then $x < y$ must be satisfied.
2) Deadline constraint: $\forall o^{(i)} \in g$, then $t^{(i)} + t_r^{(i)} + T(L^{(i)}) < \tau^{(i)}$ must be satisfied.
3) Capacity constraint: at any time, the number of riders in the vehicle cannot exceeds its capacity.

The penalty in the objective function 2 indicates dissatisfaction after the rider has waited for a long time but not been served. Note that riders in order $o^{(i)}$ can wait for a maximum response time $\max t_r^{(i)} = \tau^{(i)} - t^{(i)} - cost(l_p^{(i)}, l_d^{(i)})$, since if the response time is longer than $\max t_r^{(i)}$ its deadline constraint must be violated. In order to keep consistency with served orders, we set the penalty as the maximum response time $p^{(i)} = \max t_r^{(i)}$.

C. Hardness

We prove that the METRS problem is NP-hard by a reduction from the Shared-Route Planning Query (SRPQ) prblem [2], which has been proved as an existing NP-hard problem.

Theorem II.1. *(hardness of the METRS problem) The METRS problem defined in Definition 2 is NP-hard.*

Proof. We prove the theorem by a reduction from the SRPQ problem defined in [2], which has been proved to be an NP-hard problem. The goal of SRPQ problem is to find, for each worker $w \in W$, a route S_w, such that the total revenue of the platform $OBJ(W, O) = \sum_{w \in W} \sum_{o \in G_w} p_r$ is maximized, where p_r is the fare/payment of each order. We can rewrite the objective function of SRPQ as below:

$$\max OBJ(W, O) \Rightarrow \max \sum_{o^{(i)} \in O^+} p_r \Rightarrow \max \sum_{o^{(i)} \in O} p_r - \sum_{o^{(j)} \in O^-} p_r$$

Due to $\sum_{o^{(i)} \in O} p_r$ is a constant, we can reduce the objective function SRPQ problem into: $\min \sum_{o^{(j)} \in O^-} p_r$

Then, by setting the coefficient $\alpha = \beta = 0$ in t_e, and setting $p^{(j)} = p_r$, we show that the reduced SRPQ problem is equivalent to the METRS problem. That is, for a given SRPQ problem, we can reduce it into an instance of METRS problem. The SRPQ problem can be solved in polynomial time if and only if the METRS problem can be solved in polynomial time. Since the SRPQ problem has been proved to be NP-hard, METRS problem is also NP-hard. □

III. Overview of WATTER Framework

We first introduce the three major parts of our WATTER framework: the order pooling management algorithm, the average extra time threshold-based grouping strategy and the estimation of value function for MDP reinforcement learning [13], [14], [15] stage, as shown in Figure 2. During the online phase, we utilize a temporal shareability graph to dynamically manage temporal shareability relationships of orders. We conducted a theoretical analysis of the METRS problem to derive an average extra time threshold-based grouping strategy that utilizes the average extra time of each order and the customized threshold to make decisions. In the offline phase, we employ a MDP approach to estimate the value function, which is used as the threshold in the online decision-making process.

(a) The Graph-based Order Pooling Management. Both online and batch approaches have the disadvantage of dispatching orders too quickly. Consequently, the matching pool is restricted to the currently available orders, ignoring possible future opportunities. To tackle these issues, we propose an order pooling management algorithm based on the temporal shareability graph that serves as a data structure to maintain a dynamic set of orders: each order is represented as a node, with edges connecting it to other orders that can be shared. This part involves maintaining the graph, identifying shareable order groups, and assign workers to order groups.

(b) The Average Extra Time Threshold-based Grouping Strategy. A simple but effective strategy is to dispatch an order when the extra time is below an expected threshold θ. In reality, the expected threshold reflects the benefit that can be obtained from dispatching in the current spatiotemporal environment. We notice that a smaller threshold can lead to higher optimization results. However, a small threshold also prevents the order being grouped with more suitable orders. In Section V, we analyze that through adjusting the expected threshold θ, then we can obtain different optimization results. We transform the original METRS objective into a function of the expected threshold θ, then we solve METRS by adjusting the expected threshold for each order. Fortunately, we prove that the reduced optimization objective possesses a convex

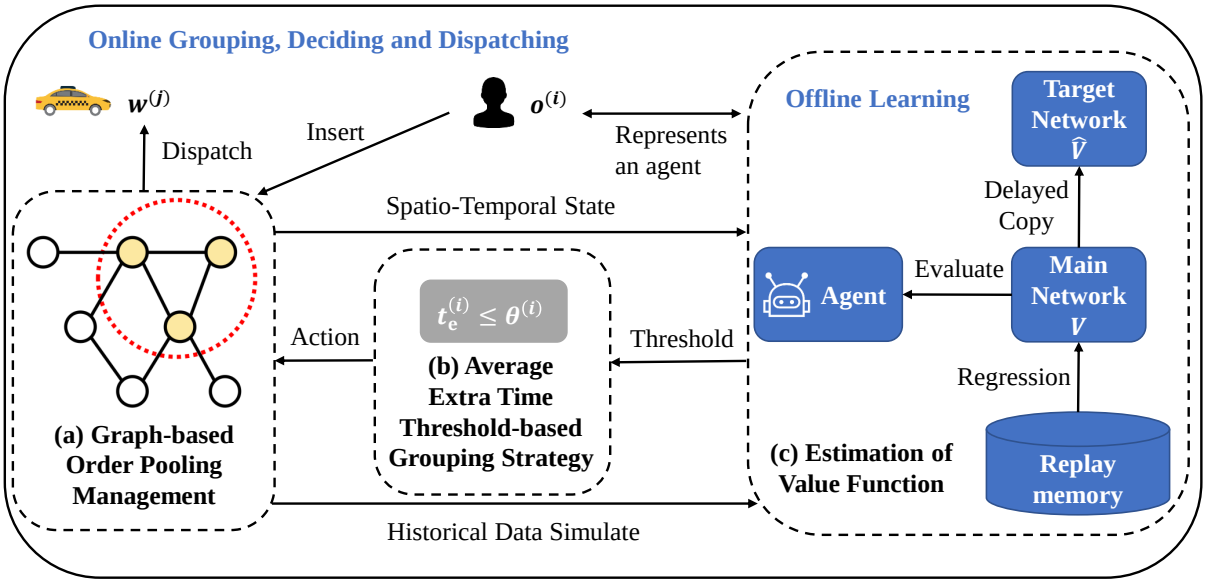


Fig. 2: Illustration of the WATTER framework.

shape, which is advantageous for solving. Then, we employ the gradient descent method to efficiently find the optimal threshold of θ.

(c) The Estimation of Value Function. To reduce the optimization objective into a function on the expected threshold θ, we previously utilized the distribution of extra times. However, the distribution may be impacted by spatiotemporal circumstances, such as peak periods, traffic congestion, supply of workers, and the demand for orders. Given historical data, we can estimate the optimal expected threshold θ [16], [15]. We formulate the dispatch decision-making process as a Markov Decision Process (MDP), treating each order as an agent. Offline learning is achieved using components such as the main network, target network, and replay memory. However, the inherent learning process often faces challenges in converging effectively due to a lack of high quality learning experiences. To overcome this, we employ an off-policy training strategy that utilizes a threshold-based strategy to generate learning experiences and minimize the disparity between the result of MDP value function and the optimal expected threshold value. It allows us to fine-tune the threshold-based strategy based on different spatiotemporal environments, ultimately determining the desired expected threshold.

IV. Graph-Based Order Pooling Management

To accommodate the dynamic arrival and departure of orders, we introduce a structure known as the temporal shareability graph.

A. Temporal Shareability Graph

Definition 8. (Temporal Shareability Graph) Given an order set O, the temporal shareability graph can be denoted by $\mathcal{G} = (O, E)$. Each $o^{(i)} \in O$ represents a node in the graph, and the edge $e = (o^{(i)}, o^{(j)}, \tau_e) \in E$ represents that node $o^{(i)}$ and node $o^{(j)}$ can be shared in a group before timestamp τ_e.

The shareability graph is dynamically updated with the arrival and departure of orders. Nodes and corresponding edges in $\mathcal{G}$ are inserted or deleted accordingly. When a new order $o^{(i)}$ arrives, it is inserted as a new node. Then, we traverse $\mathcal{G}$ to find its neighbors that can be shared. When a order $o^{(j)}$ can share with $o^{(i)}$, we can find a feasible route L with the minimal travel cost that can serve them together. A new edge $e = (o^{(i)}, o^{(j)})$ is then inserted to denote the shareability relationship between the two orders. When an order is dispatched or rejected, the corresponding node and edges are removed from $\mathcal{G}$.

At any given timestamp, the snapshot of a graph provides the current shareability relationships among orders. Clique [17] is a widely used cohesive subgraph structure for network analysis. A k-clique is a dense subgraph that has k nodes, and where each pair of nodes are adjacent. Existing studies [18], [19] have demonstrated that the sharability relationship is closely associated with the graph structure, as stated in Theorem IV.1. Thus we can efficiently enumerate shareable groups by applying clique listing algorithms [20] on the shareability graph.

Theorem IV.1. *Given a group g containing k orders, a feasible route L can be generated only if the nodes corresponding to these k orders in the sharability graph form a k-clique.*

According to Theorem IV.1, the shareable groups in the graph form cliques. The existing studies solely process the graph as a snapshot. However, our objective is to enable the graph to efficiently return the current k-cliques for decision-making. Thus, in this paper, we consider a *temporal* shareability graph, which can supports insertions/deletions of nodes, and the expiration of edges. Given a group g and a generated feasible route L, we use the τ_g to denote expiration time of the group, which is equal to the minimum slack time.

$$\tau_g = \min_{o^{(i)} \in g} \tau^{(i)} - t^{(i)} - T(L^{(i)}) - t_r^{(i)} \quad (3)$$

Due to the temporal nature of the graph, this group undergoes frequent updates. Thus, we propose an order pooling management algorithm to effectively maintain the temporal shareability graph, accommodating updates caused by order arrivals, departures, and the expiration of edges.

B. The Order Pooling Management Algorithm

To handle the online process of orders arrival, we introduce an order pooling management algorithm that operates in real-time to process orders. The orders are firstly inserted into the order pool, which is maintained as a temporal shareability graph. Periodically, we check the current status of orders and dispatch them according to a specific strategy. If the order is not dispatched, it will stay in the pool, awaiting better grouping opportunities.

As shown in Algorithm 1, we iteratively process new orders and insert them into the pool (lines 2-4). We then remove any edges and groups that will expire after the current timestamp of system (lines 5-6). Next, we go through all orders in the pool and retrieve the best group in the current spatiotemporal environment. Since we maintain a map of best group G_b during pool updates, the time cost of retrieval operation is $O(1)$ (lines 8-9). Each order in the order pool has its own waiting time, but can also leave the pool early if certain conditions (e.g. Algorithm 2) are met (line 10). *Note that here we use asynchronous periodic checks, which are performed periodically instead of after every insertion.* If we can find a valid group g, we assign it to the closest available worker (lines 11-13). If $o^{(i)}$ does not have a shareable partner at the moment, it will remain in the pool and wait. If it exceeds the wait time limit (i.e. *IsTimeout* is true), we reject it (lines 14-16).

In order to facilitate quick response of decision-maker, it is important to store the information of the current best group of each order in the pool. Each group corresponds to a k-clique in the shareability graph. When the graph is updated, pre-existing best groups may also update. There are four situations that can lead to updates of the graph and current best group: (1) order arrival (line 3); (2) order departure (line 12, 15); (3) relationship expiration (line 6); and (4) group expiration (line 6). Due to the space limitation, please refer to Appendix A and B of our technical report [21] for the detail algorithms to handle the four situations.

With the order pool, we can enable customized waiting times on the platform. A decision-maker is crucially needed to determine whether the current best group is sufficient for dispatch. For example, the online strategy is to dispatch orders as early as possible to notify users in the shortest time; the timeout strategy is to dispatch orders as late as possible to obtain better group quality.

Algorithm 1: Order Pooling Management Algorithm

Input: A set W of m workers, a set O of n orders ordered by arriving time
Output: The served groups $\mathbb{S}$ and the failed orders $\mathbb{F}$

1 initialize shareability graph $\mathcal{G}$ and best group map G_b
2 **foreach** *new order* $o^{(i)} \in O$ **do**
3 insert $o^{(i)}$ into the pool
4 update system current timestamp $t_s \leftarrow t^{(i)}$
5 **foreach** g, e *that exipres after* t_s **do**
6 remove orders in g, e from the graph
7 /* Asynchronous Periodicity Check Orders */
8 **foreach** $o^{(j)} \in \mathcal{G}$ **do**
9 $g \leftarrow G_b[j]$
10 **if** g *exists* ***and*** `MakeDecision(`g, $t^{(i)}$`)` **then**
11 assign the g to a worker to serve.
12 remove orders in group g from the graph
13 S.append(g)
14 **else if** `IsTimeout(`$o^{(j)}$`)` **then**
15 remove order $o^{(j)}$ from the graph
16 F.append($o^{(j)}$)
17 **return** $\mathbb{S}, \mathbb{F}$

V. Average Extra Time Threshold-based Grouping Strategy

A. Threshold-based Strategy

As we discussed in the introduction section, existing solutions respond orders immediately or in a static mini-batch time will prevent the riders from grouped with potential more suitable riders in a short future. Then, the platform may miss some good chances to reduce the extra times of riders. However, if the platforms hold orders for a too long time, they may get timeout. Thus, *a smart decision strategy to hold or dispatch the orders is the key component of the platform, which is also the most challenging part of the METRS problem*.

To solve the decision problem, we can examine several case studies. For orders whose current best group is optimal, we can dispatch them. For orders that are difficult to group with others, we also need to dispatch them immediately, even if their current group quality is not optimal. Orders located in popular areas can continue to wait until better group results are available. Thus, whether to continue holding or dispatching immediately depends on the spatiotemporal environment of the orders, including their pick-up and drop-off locations, as well as their release timestamps. We can use historical data to estimate the possible grouping results and calculate the expected threshold $\theta^{(i)}$ of extra time for each order $o^{(i)}$. We will introduce how to select a good $\theta^{(i)}$ for each order in Sections V-C and VI. The threshold $\theta^{(i)}$ can be considered as a reference: when the extra time $t_e^{(i)}$ of $o^{(i)}$ in a group arrangement is smaller than $\theta^{(i)}$, it means the group is better than the historical performance and $o^{(i)}$ should be dispatch with no more wait; otherwise, the order $o^{(i)}$ may wait for a better group in future.

Algorithm 2: Average Extra Time Threshold-based Grouping Strategy

Input: An order group g, system current timestamp t_s
Output: whether dispatching (True) or holding (False)

1 $t^{(i)} + \eta^{(i)} \leftarrow$ the earliest timeout time of orders in g
2 **if** $t_s > t^{(i)} + \eta^{(i)}$ **then**
3 **return** *True*
4 $\bar{t_e} \leftarrow$ average extra time of order in g
5 $\bar{\theta} \leftarrow$ average expected threshold of orders in g
6 **return** $\bar{t_e} \leq \bar{\theta}$

Based on the thresholds, we propose a flexible decision strategy for holding or dispatching orders. As shown in Algorithm 2, we filter out the orders wait longer than the limit $\eta^{(i)}$ (lines 1-3). Those orders can be served when there are suitable workers, otherwise will be rejected. We use a strategy based on expected threshold to decide whether to dispatch. First, we calculate the average extra time of orders in group g (line 4). Then, we use the average estimated threshold $\bar{\theta}$ as a reference to make the final decision (lines 5-6).

B. Reduction of Problem

In this section, we present a probability analysis of extra time. Our strategy is influenced not only by past orders, but also by the arrival probability of future ones. Therefore, we introduce the expectation of the original optimization objective $\mathbb{E}(\Phi(W, O))$ as our optimization goal, following the setting in many existing learning-based methods [13], [15], [22].

For the objective function of the METRS problem $\Phi(W, O)$, its former part $\sum_{o^{(i)} \in O^+} t_e^{(i)}$ is the extra time for all served orders; its latter part $\sum_{o^{(j)} \in O^-} p^{(j)}$ is the penalty received due to rejected orders. Note that $t_e^{(i)} \leq p^{(i)}$ holds for all orders $o^{(i)}$. To minimize $\Phi(W, O)$, we need to find a suitable time to dispatch orders so that $t_e^{(i)}$ is as small as possible and the service rate $\mu = \frac{|O^+|}{|O|}$ of the platform is as high as possible.

For each served order $o^{(i)}$, we introduce an indicator function $\mathbb{I}(i)$: $\mathbb{I}(i) = 1$ means the group of $o^{(i)}$ has average extra time smaller than its average expected threshold, that is $\bar{t_e} \leq \bar{\theta}$; $\mathbb{I}(i) = 0$ means $o^{(i)}$ has waited more than its limit wait time $\eta^{(i)}$. Thus, $O^+ = O^1 \cup O^0$, where O^1 is the set of the served orders with $\mathbb{I}(i) = 1$ and O^0 is for the served orders with $\mathbb{I}(i) = 0$. Then, the optimization objective can be rewritten as follows:

$$
\begin{aligned}
&\min \mathbb{E}\Big(\sum_{o^{(i)} \in O^+} t_e^{(i)} + \sum_{o^{(j)} \in O^-} p^{(j)}\Big) \\
\Rightarrow &\min \mathbb{E}\Big(\sum_{o^{(i)} \in O^1} t_e^{(i)} + \sum_{o^{(k)} \in O^0} (t_d^{(k)} + \eta^{(k)}) + \sum_{o^{(j)} \in O^-} p^{(j)}\Big) \\
\leq &\min \mathbb{E}\Big(\sum_{o^{(i)} \in O^1} t_e^{(i)} + \sum_{o^{(k)} \in O^0} p^{(k)} + \sum_{o^{(j)} \in O^-} p^{(j)}\Big) \\
\Rightarrow &\min \mathbb{E}\Big(\sum_{o^{(i)} \in O} \mathbb{I}(i) t_e^{(i)} + (1 - \mathbb{I}(i)) p^{(i)}\Big)
\end{aligned} \tag{4}
$$

When the order $o^{(i)}$ is positively dispatched (i.e., $\mathbb{I}(i) = 0$), the platform incurs a loss of $t_e^{(i)}$, otherwise the platform at most incurs a loss of $p^{(i)}$ if $o^{(i)}$ waits more than limit or is rejected. Therefore, the optimization objective is to minimize the total loss incurred by all orders.

Intuitively, if we minimize the extra time $t_e^{(i)}$ for each served order, we can minimize $\Phi(W, O)$. By utilizing Algorithm 2, we have:

$$
\sum_{o^{(i)} \in g} t_e^{(i)} \leq \sum_{o^{(i)} \in g} \theta^{(i)} = \bar{\theta} |g| \tag{5}
$$

where $\theta^{(i)} \in \Theta$ represents the expected threshold that we have selected for each order. We use $\bar{\theta}$ to denote average expected threshold for all orders in the current best shareable group. The extra time $t_e^{(i)}$ may change as new orders join or old orders leave the pool. It cannot be directly constrained. However, because all orders are dispatched according to the dispatch strategy, we can establish an upper bound on the original optimization problem:

$$
\begin{aligned}
&\min \mathbb{E}\Big(\sum_{o^{(i)} \in O} \mathbb{I}(i) t_e^{(i)} + (1 - \mathbb{I}(i)) p^{(i)}\Big) \\
\leq &\min_{\Theta} \mathbb{E}\Big(\sum_{o^{(i)} \in O} \mathbb{I}(i) \theta^{(i)} + (1 - \mathbb{I}(i)) p^{(i)}\Big)
\end{aligned} \tag{6}
$$

The reduced problem is a function of $\theta^{(i)}$ and $p^{(i)}$, where $p^{(i)}$ is determined by the order information and can be considered constant. Thus, the only variable we have is $\theta^{(i)}$. Under our strategy, larger threshold $\theta^{(i)}$ increases the probability of satisfying the decision condition and dispatching the order. We use $p(\mathbb{I}(i) = 1)$ to denote the probability of the indicator function being 1. Let $f(x)$ be the probability density function of the distribution that $t_e^{(i)}$ follows, and let $F(x)$ be its cumulative distribution function. Then, we have:

$$
p(\mathbb{I}(i) = 1) = \int_0^{\theta^{(i)}} f(x) dx = F(\theta^{(i)}), \tag{7}
$$

where $F(\theta^{(i)})$ measures the probability of each order $o^{(i)}$ being dispatched when its group's average extra time is smaller than its average expected threshold.

Then, the problem can be rewritten as:

$$
\begin{aligned}
&\min_{\Theta} \mathbb{E}\Big(\sum_{o^{(i)} \in O} \mathbb{I}(i) \theta^{(i)} + (1 - \mathbb{I}(i)) p^{(i)}\Big) \\
\Rightarrow &\min_{\Theta} \sum_{o^{(i)} \in O} p(\mathbb{I}(i) = 1) \theta^{(i)} + (1 - p(\mathbb{I}(i) = 1)) p^{(i)} \\
\Rightarrow &\min_{\Theta} \sum_{o^{(i)} \in O} F(\theta^{(i)}) \theta^{(i)} + (1 - F(\theta^{(i)})) p^{(i)} \\
\Rightarrow &\min_{\Theta} \sum_{o^{(i)} \in O} p^{(i)} - (p^{(i)} - \theta^{(i)}) F(\theta^{(i)}) \\
\Rightarrow &\max_{\Theta} \sum_{o^{(i)} \in O} (p^{(i)} - \theta^{(i)}) F(\theta^{(i)})
\end{aligned} \tag{8}
$$

Here, $p^{(i)} - \theta^{(i)}$ represents the minimum gain in the loss space that the platform can obtain after dispatching order $o^{(i)}$, which is a monotonically decreasing function of $\theta^{(i)}$. By setting $p^{(i)} = \tau^{(i)} - cost(s^{(i)}, e^{(i)})$, then $p^{(i)} - \theta^{(i)}$ denotes the slack time of order in the group. A longer slack time means the order is dispatched to a more appropriate group in a shorter

time. Since $F(\theta^{(i)})$ is a cumulative distribution function, it is monotonically increasing. Therefore, the product of these two functions must have a maximum value. In other words, the reduced objective function is a convex function of thresholds of orders. Our goal is to find the threshold $\theta^{(i)}$ that corresponds to the maximum value of this product function.

C. Distribution Fitting and Optimization

As mentioned before, we treat the extra time t_e as a random variable that follows a specific distribution. We then transform the optimization objective into a function of the expected threshold θ. There are two key challenges: (1) determining the specific distribution of t_e, and (2) optimizing the objective function under the distribution assumption.

Distribution Fitting. Regarding the extra time of orders, we have several observations about its distribution: (1) Shorter orders may exhibit smaller extra time due to the difficulty of inserting detours along their routes; (2) Orders with both the pick-up and drop-off locations in popular areas may experience smaller extra time because there are more opportunities for sharing with similar orders; (3) Orders released during peak hours may result in smaller extra time as a large number of proper orders can grouped together. Thus, we can reasonably assume that the extra time of orders can be clustered, with many orders falling into the same sub-intervals. However, obtaining prior knowledge about the specific distributions for these clusters and the confidence level with which they adhere to these distributions is challenging.

For random variables influenced by multiple factors, Gaussian Mixture Models (GMM) [23] are commonly used. GMM incorporates multiple Gaussian distributions and combines them based on weights, making it suitable for fitting complex distributions. In the case of the extra time t_e, we can consider each influencing factor mentioned earlier as a sub-component of the GMM, collectively contributing to the overall distribution. The fitting of GMM can be accomplished using the Expectation-Maximization (EM) algorithm [24], which is widely employed for this purpose.

Objective Optimization. Our reduced optimization objective is a convex function of the thresholds $\theta^{(i)}$ of orders $o^{(i)}$, ensuring the existence of a maximum value. For convex functions and approximate convex functions, various methods such as Newton's method or gradient descent can be employed to determine the optimal threshold. While these methods may encounter local optima, for the optimization objective in this paper, only a few iterations are required to obtain the solution.

As shown in Algorithm 3, we firstly utilize historical data of the extra time as the distribution to be fitted. We then employ the EM algorithm to fit a Gaussian Mixture Model to this historical data (line 1). After obtaining the fitted distribution, we can easily calculate its cumulative distribution function, denoted as $F()$ (line 2). For each individual order $o^{(i)}$, we can calculate the function $g(\theta^{(i)}) = (p^{(i)} - \theta^{(i)})$ based on its penalty term (line 3-4). By combining these two functions, we can use the gradient descent method to find the optimal expected threshold $\theta^{(i)}$ (line 5-6).

Algorithm 3: Distribution Fitting and Optimization

Input: order set O, historical records of extra time H
Output: optimal expected thresholds Θ

1 $M \leftarrow$ the GMM fitting result on H
2 $F \leftarrow$ the CDF of M
3 **foreach** $o^{(i)} \in O$ **do**
4 $\quad g(\theta^{(i)}) = (p^{(i)} - \theta^{(i)})$
5 $\quad \theta^{(i)} \leftarrow$ gradient descent result on $F() * g()$
6 $\quad \Theta \leftarrow \Theta \cup \theta^{(i)}$
7 **return** Θ

To summarize, we reduce the METRS problem to maximize a function of $\theta^{(i)}$. The core challenge of the reduced problem is estimating the expected threshold $\theta^{(i)}$ for each order $o^{(i)}$. Then, we utilize a Gaussian Mixture Model for distribution fitting and employ gradient descent to optimize the reduced objective. However, in practical situations, the optimal group of orders is usually dispatched quickly, then leads to no enough samples to accurately fit its distribution. While we can make assumptions about the distribution of t_e, the obtained solution is only a *coarse-grained* result. The dispatch strategy is still heavily influenced by the spatiotemporal environment. Moreover, it is difficult to get the distribution information of new arrived orders in the future.

VI. Reinforcement Learning based Estimation

To overcome the shortcomings in Section V-C, we can estimate the expected threshold based on a reinforcement learning approach *offline* with historical data. Combined with the *online* maintenance of the current best group, the dispatch strategy can be *fine-grained* tailored for each order.

A. Dispatch Strategy as MDP

Orders in the pool must be decided on whether to dispatch them or not. If they are not dispatched, the orders wait in the pool and go through multiple decisions until they are either dispatched or expire. In this paper, we propose to model this process as a Markov Decision Process (MDP) from a local view, where each individual order is modeled as an agent. The MDP captures the sequential decision process as an agent observes the current *environment*, takes an *action* a, and *transits* from state s_t to $s_{t+\Delta t}$, then receives a certain *reward* r. Here use t as the subscript of the state to denote the time that the order is waited, such that s_0 representing the initial state.

State. Follow the design in study [22], we consider multiple spatio-temporal features in state s_t. These spatio-temporal features consist of two components: the basic feature and the environmental feature. The basic feature contains information such as the order's pick-up location and release timestamp. We use *location index* and *time index* to quantize the basic spatio-temporal feature of the order into and a number of regions and timeslots. The location index is obtained by dividing the examined city area into $n \times n$ region grids, which is commonly used in existing studies [22], [25]. The time index is obtained

by dividing the time into intervals of Δt seconds. The region information of the pick-up and drop-off locations of the order is represented as a vector s_L using one-hot encoding. The timeslots of the order's release and wait are connected and fed into a two-dimensional vector s_T.

The environmental features include the current demand and supply distribution of the platform. We consider the current demand distribution of existing orders on the platform, including the distribution of pick-up locations and drop-off locations represented by vector s_O. The supply distribution of workers in each region is represented using vector s_W. The distribution vectors are all calculated by the location index. Combining this data, the spatiotemporal environmental state can be written as $s_t = [s_L, s_T, s_O, s_W]$.

Action. For each decision phase, there are two types of actions that an agent can perform. The *dispatch* action with $a = 1$ involves grouping the order and finding a worker to serve it. This means that the agent considers that the current order has matched the desired group and can leave the pool. Another action is *wait* with $a = 0$. This action is crucial for our problem since the order may be waiting in the pool for a longer time, but it's still under our optimization consideration. Specifically, the wait action means the agent thinks it will be matched to a better group in the future.

State Transition. The sequential decision process involves waiting actions during the life cycle of an order. A wait action triggers a transition to the next state with the same location but a different timeslot and environment. In the case of dispatch or expiration, the order will terminate its life cycle and receive a final reward. The dispatch action assigns the order to a worker along with the current best group, while expiration occurs in situations where the order has been waiting in the pool for too long. Expiration occurs implicitly in the sequential decision process and is unobservable to the agent. Since the rider becomes more impatient, the order may be cancelled at any time.

Reward. The definition of reward r_t depends on the optimization objective of the platform, which in this paper is transformed into Equation 8. Let T be the time order to be dispatched or rejected. The key objective to solve MDP is to learn the *state value function* $V_\pi(s_t) = E(\sum_{i=1}^{T-t} \gamma^{i-1} r_{t+i})$ under the current strategy π, where γ is the discount factor that controls how far the agent looks into the future for rewards. By using MDP to model the dispatch decision, the expected value is corresponds to the "state-value function". The strategy π used here is Equation 5. That is, $\theta^{(i)} = V_\pi(s_t^{(i)})$, where $s_t^{(i)}$ is the spatio-temporal environment of the order $o^{(i)}$. The Bellman Update corresponding to each of the two different actions are

$$V(s_t^{(i)}) \leftarrow \begin{cases} p^{(i)} - t_d^{(i)} & a = 1 \\ -\Delta t + \gamma^{\Delta t} V(s_{t+\Delta t}^{(i)})(1 - \mathbb{I}(expired)) & a = 0 \end{cases} \tag{9}$$

Note that the wait action may result in two types of rewards: (1) continuing to wait in the next round, which has an immediate reward of $-\Delta t$. Only in this situation is the state-value function related to the future state and reward. (2) being expired, which has a reward of 0 and the corresponding indicator $\mathbb{I}(expired) = 1$. When the agent takes the dispatch action, it will receive a final positive reward and a negative reward for the detour time of the order in the current best group.

Although we set an immediate reward for each action, the actual reward accumulates over each decision phase. It can be calculated as $\sum_{t=0}^{t_r^{(i)}/\Delta t} -\gamma^t \Delta t$. By setting the discount factor $\gamma = 1$, the reward accumulated over the decision phases is equal to the negative waiting response time $-t_r^{(i)}$. The accumulated reward R can be calculated as follows:

$$R(s_0^{(i)}) = \begin{cases} -t_r^{(i)} + p^{(i)} - t_d^{(i)} = p^{(i)} - t_e^{(i)} & dispathed \\ -t_r^{(i)} = -\max t_r^{(i)} & expired \end{cases} \tag{10}$$

where $s_0^{(i)}$ is the initial state of agent representing order $o^{(i)}$.

B. Deep-Q-Network Learning Approach

The aim of MDP is to maximize the expectation of the accumulated reward for each agent. If an agent takes too many wait actions, it may cause the expiration of an order and result in a negative reward. A well-trained agent will prevent this from happening. When dealing with whether to dispatch, the agent will carefully consider whether a wait action is needed in each decision phase to obtain a higher cumulative reward. Therefore, the total benefit of MDP is consistent with our optimization objective Equation 8.

We use the state-value function as an estimation of the expected value in Equation 5 to decide whether to dispatch and ask agents to get the maximum benefit during the training phase. As a result, the learned state-value function is the optimal expected threshold θ. To achieve this, we use existing value-based methods, such as *Q*-Learning [26], [27] and Deep *Q*-Networks (DQN) [28], [22], and we use a *replay memory* M to store the experience tuples. There are two networks: (1) the *main network* V , used to estimate the value function; and (2) the *target network* $\hat{V}$, which is a delayed copy of V and is used to stabilize the training process. Mean-squared Temporal-Difference (TD) Error is a commonly used loss function to estimate the value function:

$$loss_{td}(s) = (r_t + \gamma \hat{V}(s_{t+\Delta t}) - V(s_t))^2 \tag{11}$$

After each action is taken, we update the value function V by taking a gradient descent to minimize loss function L. However, the TD loss alone is insufficient to meet our expectations for estimating the expected threshold. This is because the TD loss only guarantees the value relationship between states. We also require the value function to have a relationship with the true extra time t_e so that it can be directly utilized in the threshold-based strategy. Therefore, we introduce the target loss to learn the threshold.

$$loss_{tg}(s) = (p^{(i)} - \theta^{(i)} - V(s_t))^2 \tag{12}$$

where $\theta^{(i)}$ is the optimal threshold obtained through distribution fitting and optimization in Section V.

The TD loss guides the agent in selecting the action, either waiting or dispatching, that has a higher value. This ultimately helps the value function approximate the true higher value.

TABLE II: Experimental Settings.

Parameters	Values
the number m of riders	(NYC) 50K, 75K,**100K**, 125K
	(CDC, XIA) 30K, **40K**, 50K, 60K
the number n of vehicles	3K, 4K, **5K**, 6K
the deadline $\gamma(\times cost(s^{(i)}, d^{(i)}))$	1.2, 1.4, **1.6**, 1.8
the maximum capacity of vehicles K_w	2, **3**, 4, 5
the balance parameter α and β	**1**

Meanwhile, the target loss aims to align the DQN's learning outcomes with the existing strategy, allowing for fine-tuning through actions based on the established policy. Consequently, our final loss is a weighted sum of these two parts.

$$loss_{(s_t,a,s_{t+\Delta t},r_t)\in M}(s) = \omega loss_{td} + (1-\omega) loss_{tg}; \quad (13)$$

where ω is the weight parameter.

Note that there can be multiple implementation approaches for the experience generation in the trainning process. (1) the agent can make dispatch decisions based on the value function. (2) the value function can be used as an estimation of threshold. The first approach has a limitation in that the difference between the current state and the next state may not be significant. If the neural network is not deep enough, it may struggle to distinguish between the two states, leading to a policy with high randomness. Additionally, considering the delay, we do not employ large-scale neural networks as the implementation of the value function. The second approach has a drawback in the initial training phase, where parameter initialization can bias the value function towards smaller expected thresholds, making it challenging to obtain high-quality training experiences. Therefore, we adopt an off-policy approach to enhance the training process. In the initial training phase, we collect training experiences using the threshold-based strategy and the optimal expected threshold from Section V. In the later stage, we utilize the second approach to gather training experiences for fine-tune. (see Appendix C of our technical report [21] for more detailed algorithms)

VII. Experimental Study

In this section, we present the experimental setup and results of our algorithm.

A. Experimental Setup

Data Set. We evaluate our algorithm on three real-world city datasets. The first is a public dataset collected from yellow taxis [29] in New York City (noted as NYC), USA. The second and third are order datasets collected from the GAIA platform of Didi Chuxing [30] in Chengdu (noted as CDC) and Xi'an (noted as XIA), China. We use order data from 1 to 31 July, 2013 for NYC, from 1 to 29 November, 2016 for CDC and from 1 to 30 October 2016 for XIA to train the value function proposed in Section V. In the experiments for evaluating parameters, we use order data from 07 July, 2013 in New York, 31 November, 2016 in Chengdu and 31 October 2016 in Xi'an. Each order in the dataset contains the longitude and latitude of the pick-up and drop-off locations as well as the time of the order. The specific experiment-related parameters are shown in Table II (the bold part is the default parameter).

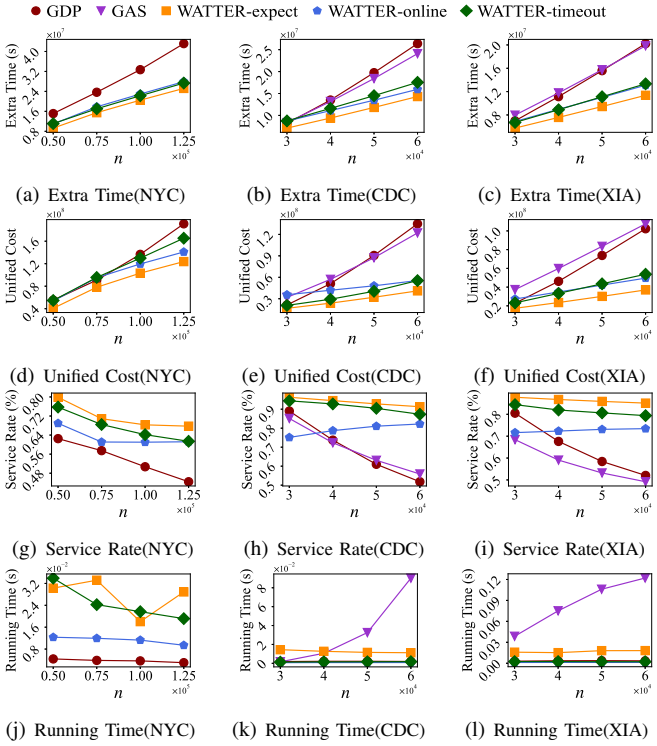


Fig. 3: Performance of varing n.

Implementation. We simulate ridesharing in our framework using the following settings. Follow the grid index construction methods in existing studies [1], [22], [25], we partition the city into several cells to serve as a grid index to speed up workers and riders search. We tested the performance impact of different grid size and choose a grid of 10×10 cells as the setting of grid index (For a detailed analysis, please refer to Appendix D of our technical report [21]). We treat each record as an order with one passenger and thus a k-clique in our framework can represent a group of orders with k riders. In addition, we set the deadline for each order (e.g. $\tau^{(i)} = t^{(i)} + \gamma * cost(s^{(i)}, d^{(i)})$), which is a frequently used setup in numerous previous studies [10], [4], [2]. To achieve fairness, we generate the maximum passenger capacity parameter K_w and the pick-up position for workers for the three datasets based on specific distributions. We randomly generate initial positions for workers using the distribution of orders' pick-up positions. The vehicle capacity $k^{(j)}$ of worker is randomly sampled within the range $[2, K_w]$.

All experiments were implemented in C++ and compiled using -O3 optimization. The experiments were run on a single server equipped with a Xeon Silver 4214 CPU @ 2.20GHz and 128GB RAM. All algorithms are run in a single thread.

Compared Algorithms. We compare our algorithm *WATTER-expect* with the following algorithms:

- *WATTER-online.* (this paper) A variant implemented in the Order Pooling Management Algorithm that uses online strategy for dispatching orders. Each order is dispatched as early as possible.
- *WATTER-timeout.* (this paper) Another variant implemented

in the Order Pooling Management Algorithm that uses time-out strategy for dispatching orders. Each order is dispatched as late as possible.

- *GDP* [10] An online-based algorithm where orders can only use information from existing orders. It greedily tries to insert the pick-up and drop-off locations of orders into the worker's route.
- *GAS* [2] A batch-based algorithm where orders within a batch are processed together. It generates an additive tree for all orders that can be served by each worker and finds the group with the maximum utility in the tree to dispatch.

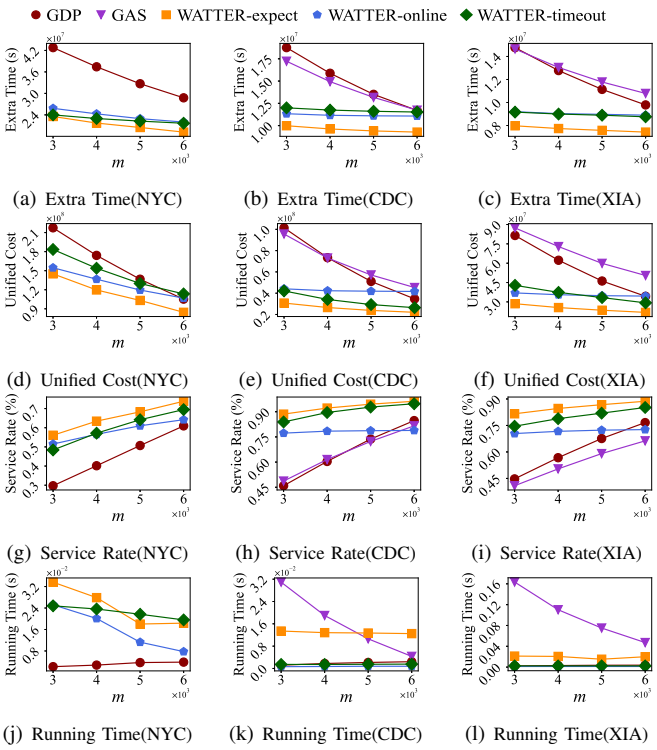
Measurements. All algorithms are evaluated in terms of *Extra Time(s)*, *Unified Cost* [10], *Service Rate(%)* ($|O^+|/|O|$) and *Running Time(s)*. The unified cost UC is calculated by the sum of worker cost and penalty for rejected orders. We set the balance parameter as 1 in UC, and the penalty is $10 \times cost(r)$. The running time is the average algorithm running time of each order. The metrics are widely used in the existing large-scale ridesharing studies [10], [2], [31] excpet for extra time, which is the optimization objective of our paper. We early terminate the algorithms that not completed experiments within 24 hours.

B. Experimental Results

Impact of Varying Number of Riders. Figure 3 presents the results of varying the number of orders. In all datasets, our proposed algorithms WATTER-expect, WATTER-online, and WATTER-timeout are more effective than existing approaches with the increase of the number of orders. However, when the number of orders is low, GDP and GAS may have slightly better results. The optimization effect of our framework is insignificant due to the lower order-worker to quantity ratio. For instance, when $n = 50k$, WATTER-expect achieved 12.2%, 18.4%, 35.7% and 40.1% lower extra time compared to WATTER-online, WATTER-timeout, GAS and GDP in the CDC dataset, respectively.

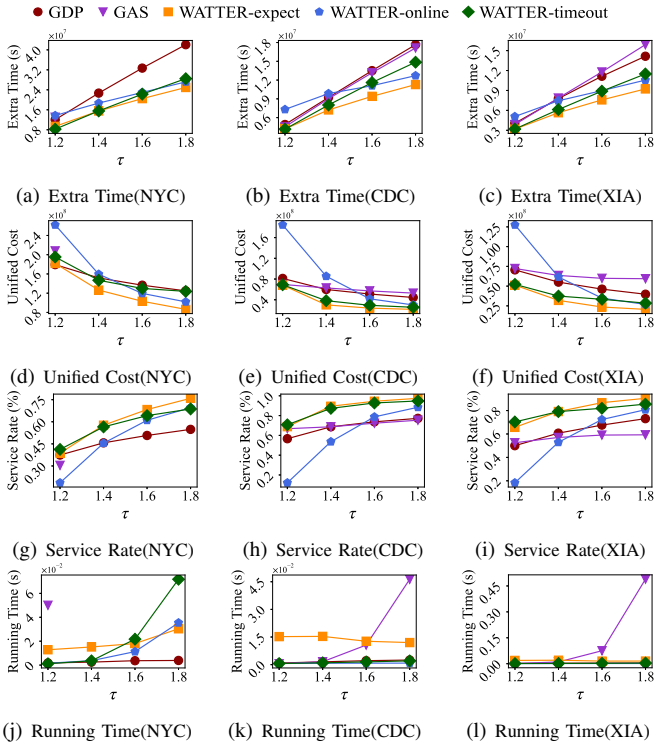
Regarding service rate, WATTER-expect is the highest, WATTER-timeout is the first runner-up, and WATTER-online is the second runner-up. This is because, with more orders, waiting for a longer time (e.g., WATTER-timeout) results in higher group quality and higher worker utilization. For instance, when $n = 50k$, WATTER-expect achieved 2.3%, 11.7%, 36.9% and 41.0% improvement in service rate compared to WATTER-timeout, WATTER-online, GAS, and GDP in the CDC dataset, respectively.

When it comes to running time, GDP is the fastest algorithm due to its greedy insertion without enumerating possible order groups. GAS, however, has an exponentially increasing time cost due to an increase in the number of orders. Among the three algorithms proposed in this paper, WATTER-online is faster than WATTER-timeout because it maintains a small enough order pool to minimize the insertion and update costs of new orders. WATTER-timeout, on the other hand, maintains the largest order pool, resulting in the higher running time. Due to the need to use neural network, the runing time of WATTER-expect is second-highest for most cases.



(a) Extra Time(NYC) (b) Extra Time(CDC) (c) Extra Time(XIA)
(d) Unified Cost(NYC) (e) Unified Cost(CDC) (f) Unified Cost(XIA)
(g) Service Rate(NYC) (h) Service Rate(CDC) (i) Service Rate(XIA)
(j) Running Time(NYC) (k) Running Time(CDC) (l) Running Time(XIA)

Fig. 4: Performance of varying m.



(a) Extra Time(NYC) (b) Extra Time(CDC) (c) Extra Time(XIA)
(d) Unified Cost(NYC) (e) Unified Cost(CDC) (f) Unified Cost(XIA)
(g) Service Rate(NYC) (h) Service Rate(CDC) (i) Service Rate(XIA)
(j) Running Time(NYC) (k) Running Time(CDC) (l) Running Time(XIA)

Fig. 5: Performance of varing τ.

Impact of Varying Number of Workers. Figure 4 presents the results of varying the number of workers. Across all datasets, our algorithm delivers the best performance in all metrics except for running time. As the number of workers increases, both the extra time and the unified cost decrease.

This is because it becomes easier to find an available worker closer to the order group, reducing the worker's response time and detour and increasing the service rate. For instance, in the NYC dataset, when $m = 6000$, WATTER-expect outperformed WATTER-timeout, WATTER-online, and GDP, achieving a 4.3%, 9.6%, and 12.8% improvement in service rate, respectively. For WATTER-timeout, the service rate gradually surpasses WATTER-online in the NYC dataset when the number of workers increases because more workers switch to idle status during riders' waiting. Notably, the performance of the WATTER-online method exhibits less variation with increasing drivers on the CDC and XIA datasets. This is attributed to the fact that the orders in these two datasets have more dispersed pick-up and drop-off locations compared to the NYC dataset, where most orders are concentrated in the Manhattan area. As a result, the main limitation of WATTER-online is the difficulty in finding suitable shareable group.

Impact of Varying Deadline Figure 5 presents the results of varying the orders' deadlines. Under small deadlines, the algorithms proposed in this paper show little difference from the baselines in terms of extra time. This is because small deadlines do not allow for orders to wait for too long. However, as the deadline increases, the WATTER-expect outperforms the baselines. For example, considering unified cost, at $\tau = 1.8$, WATTER-expect has a decrease of 23.1%, 27.7%, 48.2%, and 65.3% compared to other baselines in the XIA dataset, respectively.

While longer deadlines improve the service rate of GDP by making it easier to insert riders into workers' routes, it do not necessarily provide better match partners for riders. For GAS, the final unified cost was the highest in most cases because it did not consider the cost of workers when selecting groups. However, longer deadlines increase the extra time allowed for orders, making it more likely to find a more suitable group. Furthermore, the wait time for riders enables more workers to become reliable, enhancing the effectiveness of our algorithms. We observed that WATTER-online shows the most significant improvements across all performance metrics on varing deadline. This is because the bottleneck of WATTER-online lies in the challenge of immediately finding shareable groups, and the increase in the deadline greatly reduces this difficulty.

Impact of Varying Maximum Capacity of Workers Figure 6 presents the results of varying the workers' vehicles maximum capacity. It is observed that the algorithm WATTER-expect proposed in this paper shows superiority when the maximum capacity changes. For instance, when $K_w = 4$, WATTER-expect achieved the following reductions in extra time compared to the baselines: 15.3%, 18.9%, 23.5%, and 28.1% in CDC dataset, respectively. Increasing the maximum capacity K_w significantly decreases the extra time and unified cost for GDP. However, our algorithms show less fluctuation due to several reasons. (1) when optimizing for extra time, the performance of the group with a capacity of 2 is dominant. (2) the GDP greedily inserts workers without considering waiting time and dispatch the order as long as there is a

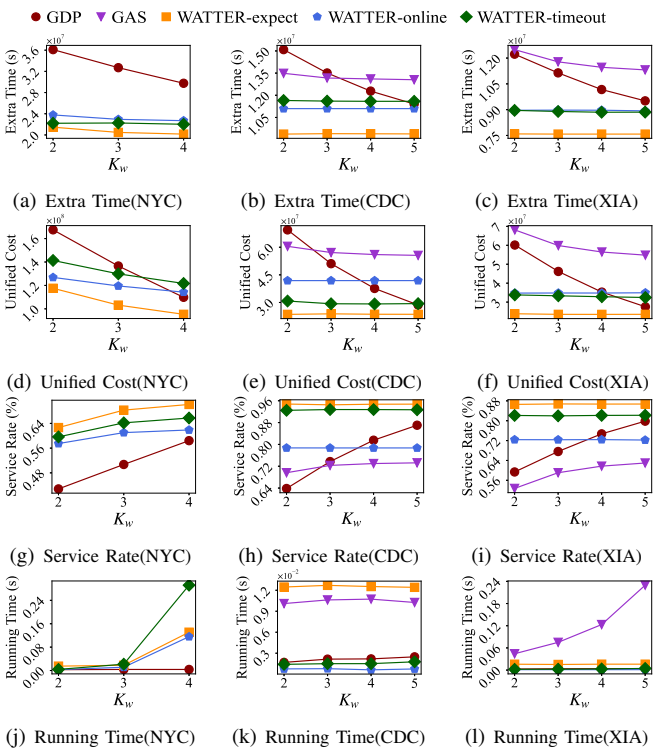


Fig. 6: Performance of varing K_w.

slot available for insertion. Increasing the capacity limit can effectively improve the service rate, further optimizing extra time and unified cost. (3) the non-preemptive mode we adopt means that workers cannot serve two groups at the same time. If the capacity of a worker's vehicle exceeds the size of the group to be served, the excess cannot be utilized efficiently. (4) the shareability graph on the CDC and XIA datasets is sparse, making it difficult to enumerate groups whose size is greater than 2. This can also be verified by the running time, as our algorithms show an increase in running time as the maximum capacity increases in the NYC dataset, but not in the CDC and XIA datasets.

Trade-off between Response and Detour Time We present the average proportions of response time and detour time for all served orders in Figure 7. The results are obtained on the CDC dataset with default parameters. We can observe that among the three variants of the WATTER algorithm, the WATTER-online achieves the lowest response time but also results in the highest detour time. On the other hand, the WATTER-timeout variant exhibits the highest response time and the lowest detour time. It support our assumptions regarding response time and detour time. That is, the longer waiting increases the likelihood of finding a higher-quality group, leading to a smaller average detour time. Meanwhile, the GDP and GAS show the same trend with WATTER-online. In contrast, WATTER-expect effectively balances response time and detour time. It achieves this balance by allowing orders to wait for an appropriate duration, effectively reducing detour time. At the same time, this waiting period is not as extreme as in WATTER-timeout, resulting in the lowest extra time.

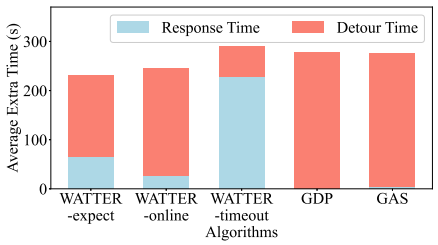


Fig. 7: Trade-off between response and detour time.

Summary of the Experimental Results We summarize the major experimental results in the following.

1) In terms of the total extra time and unified cost, our order pooling management algorithm algorithms WATTER-expect, WATTER-online and WATTER-timeout have superiority compared to the online-based method GDP and batch-based method GAS.
2) The threshold-based strategy has a noticeable effect on balancing the response time and group quality. The WATTER-expect have less extra time than WATTER-online and WATTER-timeout up to 10.6% and 18.4%.
3) GDP are more efficient than our algorithms when the orders are more similar, e.g. in the NYC dataset. It is because the shareability graph are larger and denser, which makes huge group enumeration cost.

VIII. Related Work

The ridesharing problem is a variant of the dial-a-ride problem, which involves planning the routes of workers to serve orders with specific pick-up and drop-off locations. Unlike the dial-a-ride problem, the ridesharing problem primarily focuses on the additional benefits of grouping orders together. Recent studies can be classified mainly based on processing frameworks and optimization objectives.

Process frameworks. Online-based methods use the insertion operator as a heuristic solution for planning work routes. They greedily and incrementally insert each newly arriving order into the worker's current route based on certain objectives. Zheng *et al.* [8] enumerate all possible insertion positions to search for the optimal solution. Huang *et al.* [11] proposed the structure called kinetic tree, which maintains and provides the optimal schedule for vehicles when new orders arrive. Tong *et al.* [10] proposed a dynamic programming (DP) algorithm to insert and check constraints, reducing the operation time from cubic to linear. Wang *et al.* [31] proposed demand-aware insertion based on predictions about future orders.

Batch-based methods process all orders within a mini-batch time at once. They typically enumerate all possible groups to find the optimal one and then assign the groups to workers [32], [33], [1], [2], [34]. Bei *et al.* [33] formulated the problem of combinatorial optimization between orders and workers. Cheng *et al.* [1] use machine learning models to predict future vehicle demand. They then used a queue-theoretic framework to balance the demand and supply. Zeng *et al.* [2] proposed an index called additive tree to accelerate the enumeration and greedily choose the most profitable group to serve. The effectiveness and runtime of batch-based methods depend on the batch size setting. If the batch size is large, the runtime increases exponentially. In contrast, the utility achieved is less competitive than that of online-based methods.

Optimization objectives. The primary optimization objectives for ridesharing services are from the perspectives of the platform and workers, with other objectives acting as constraints. From the platform's perspective, previous studies [1], [2], [3] have mainly focused on maximizing revenue and the number of served riders. The platform's revenue is calculated by subtracting travel cost of workers from the payment of riders. Given a fixed service rate, reducing the worker's travel cost can increase revenue.

Many studies [4], [5], [6] focus on decreasing travel costs from the worker's perspective. This can increase both the platform's revenue and the workers' earnings. To unify these objectives, Tong *et al.* [10] introduced the objective of a unified cost, which integrates the three goals into one and illustrates the relationships among them.

Regarding the rider's perspective, most existing studies [9], [35], [10] impose the time constraint that riders must be delivered before the deadline. Among these studies, the manually set deadline is an important parameter that affects the algorithms' performance. Flow time [36], [37], [4] is a commonly used objective for improving the rider satisfaction.

However, due to limitations in the processing framework, studies on ridesharing provide group results to riders either immediately or in mini-batches, without taking into account the response time. In this paper, we propose a new optimization objective for the METRS problem that balances the response time and detour time to improve riders' satisfaction. Different from existing processing frameworks, we propose a novel order pooling management algorithm to handle the balance problem, which is more effective and efficient than previous studies.

IX. Conclusion

In this paper, we study the Minimal Extra Time RideSharing (METRS) problem in which orders arrive dynamically and the platform needs to group them and assign workers to serve the groups as many as possible while maximizing rider satisfaction. We prove that the METRS problem is NP-hard. To address this challenge, we propose an efficient approach WATTER. It utilizes order pooling management algorithm to allow riders' orders to wait in a temporal shareability graph-based order pool until they can be grouped effectively. We also devise a avreage extra time threshold-based grouping strategy that determines when to dispatch the orders in the pool. To adapt to different spatio-temporal environments of orders, we model the decision process as a Markov Decision Process (MDP) and use a reinforcement learning model to solve it. We propose three algorithms, namely WATTER-expect, WATTER-online, and WATTER-timeout, and demonstrate their efficiency and effectiveness through experiments on three real datasets.

REFERENCES

[1] P. Cheng, C. Feng, L. Chen, and Z. Wang, "A queueing-theoretic framework for vehicle dispatching in dynamic car-hailing," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1622–1625, 2019.

[2] Y. Zeng, Y. Tong, Y. Song, and L. Chen, "The simpler the better: An indexing approach for shared-route planning queries," *Proc. VLDB Endow.*, vol. 13, p. 3517–3530, oct 2020.

[3] T. Wang, H. Luo, Z. Bao, and L. Duan, "Dynamic ridesharing with minimal regret: Towards an enhanced engagement among three stakeholders," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2022.

[4] Y. Xu, Y. Tong, Y. Shi, Q. Tao, K. Xu, and W. Li, "An efficient insertion operator in dynamic ridesharing services," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1022–1033, 2019.

[5] Z. Liu, Z. Gong, J. Li, and K. Wu, "Mobility-aware dynamic taxi ridesharing," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 961–972, 2020.

[6] M. Haliem, G. Mani, V. Aggarwal, and B. Bhargava, "A distributed model-free ride-sharing approach for joint matching, pricing, and dispatching using deep reinforcement learning," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 12, pp. 7931–7942, 2021.

[7] D. O. Santos and E. C. Xavier, "Dynamic taxi and ridesharing: A framework and heuristics for the optimization problem," in *Twenty-third international joint conference on artificial intelligence*, 2013.

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

[9] P. Cheng, H. Xin, and L. Chen, "Utility-aware ridesharing on road networks," in *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1197–1210, ACM, 2017.

[10] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu, "A unified approach to route planning for shared mobility," *Proc. VLDB Endow.*, vol. 11, p. 1633–1646, jul 2018.

[11] Y. Huang, F. Bastani, R. Jin, and X. S. Wang, "Large scale real-time ridesharing with service guarantee on road networks," *Proc. VLDB Endow.*, vol. 7, p. 2017–2028, oct 2014.

[12] X. Bei and S. Zhang, "Algorithms for trip-vehicle assignment in ridesharing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[13] A. O. Al-Abbasi, A. Ghosh, and V. Aggarwal, "Deeppool: Distributed model-free algorithm for ride-sharing using deep reinforcement learning," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 12, pp. 4714–4727, 2019.

[14] S. Shah, M. Lowalekar, and P. Varakantham, "Neural approximate dynamic programming for on-demand ride-pooling," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 507–515, Apr. 2020.

[15] X. Tang, F. Zhang, Z. Qin, Y. Wang, D. Shi, B. Song, Y. Tong, H. Zhu, and J. Ye, "Value function is all you need: A unified learning framework for ride hailing platforms," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 3605–3615, 2021.

[16] Z. Xu, Z. Li, Q. Guan, D. Zhang, Q. Li, J. Nan, C. Liu, W. Bian, and J. Ye, "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*, pp. 905–913, 2018.

[17] C. Zhang, Y. Zhang, W. Zhang, L. Qin, and J. Yang, "Efficient maximal spatial clique enumeration," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 878–889, 2019.

[18] Y. Chen and L. Wang, "P-ride: A shareability prediction based framework in ridesharing," *Electronics*, vol. 11, no. 7, 2022.

[19] H. Wu, Y. Chen, L. Wang, and G. Ma, "E-ride: An adaptive event-driven windowed matching framework in ridesharing," *IEEE Access*, vol. 10, pp. 43799–43811, 2022.

[20] Z. Yuan, Y. Peng, P. Cheng, L. Han, X. Lin, L. Chen, and W. Zhang, "Efficient $k-$clique listing with set intersection speedup," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pp. 1955–1968, 2022.

[21] "[online] wait to be faster: a smart pooling framework for dynamic ridesharing [technical report]." http://cspcheng.github.io/pdf/AdaptiveWindowRidesharing.pdf.

[22] J. Ke, F. Xiao, H. Yang, and J. Ye, "Learning to delay in ride-sourcing systems: A multi-agent deep reinforcement learning framework," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 5, pp. 2280–2292, 2022.

[23] C. Rasmussen, "The infinite gaussian mixture model," in *Advances in Neural Information Processing Systems* (S. Solla, T. Leen, and K. Müller, eds.), vol. 12, MIT Press, 1999.

[24] G. Xuan, W. Zhang, and P. Chai, "Em algorithms of gaussian mixture model and hidden markov model," in *Proceedings 2001 International Conference on Image Processing (Cat. No.01CH37205)*, vol. 1, pp. 145–148 vol.1, 2001.

[25] J. Jin, P. Cheng, L. Chen, X. Lin, and W. Zhang, "Gridtuner: Reinvestigate grid size selection for spatiotemporal prediction models," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pp. 1193–1205, 2022.

[26] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[27] H. Hasselt, "Double q-learning," *Advances in neural information processing systems*, vol. 23, 2010.

[28] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, p. 2094–2100, AAAI Press, 2016.

[29] "[online] TLC Trip Record Data.." https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[30] "[online] Didi Chuxing.." http://www.didichuxing.com/.

[31] J. Wang, P. Cheng, L. Zheng, C. Feng, L. Chen, X. Lin, and Z. Wang, "Demand-aware route planning for shared mobility services," *Proc. VLDB Endow.*, vol. 13, p. 979–991, mar 2020.

[32] L. Zheng, L. Chen, and J. Ye, "Order dispatch in price-aware ridesharing," *Proceedings of the VLDB Endowment*, vol. 11, no. 8, pp. 853–865, 2018.

[33] X. Bei and S. Zhang, "Algorithms for trip-vehicle assignment in ridesharing," in *AAAI Conference on Artificial Intelligence*, 2018.

[34] L. Zheng, P. Cheng, and L. Chen, "Auction-based order dispatch and pricing in ridesharing," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1034–1045, IEEE, 2019.

[35] L. Chen, Q. Zhong, X. Xiao, Y. Gao, P. Jin, and C. S. Jensen, "Price-and-time-aware dynamic ridesharing," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1061–1072, 2018.

[36] M. Firat and G. J. Woeginger, "Analysis of the dial-a-ride problem of hunsaker and savelsbergh," *Operations Research Letters*, vol. 39, no. 1, pp. 32–35, 2011.

[37] L. Häme, "An adaptive insertion algorithm for the single-vehicle dial-a-ride problem with narrow time windows," *European Journal of Operational Research*, vol. 209, no. 1, pp. 11–22, 2011.

APPENDIX

A. Order Arrival Algorithm

We illustrate the update triggered by new order arrival, as shown in Algorithm 4. To accelerate the subsequent decision-making process, we maintain a map G_b that stores the current best group information of orders. When a new order arrives, we take the following three steps: (1) Find neighbors and add edges (line 1); (2) Enumerate new k-cliques (line 2); (3) Update the map G_b based on the enumeration results (lines 3-11). In the third step, we utilize the Insertion Algorithm [10] to generate the fastest feasible route for each group. Although the neighbors may have better groups due to the arrival of new orders, but the entire insertion process can still be completed with only one enumeration.

Algorithm 4: Order Arrival

Input: order $o^{(i)}$, original graph $\mathcal{G}$, map G_b
Output: updated graph $\mathcal{G}$, updated map G_b

1 traverse the graph $\mathcal{G}$, find neighbors that can be shared with $o^{(i)}$ and add new edges
2 $G^{(i)} \rightarrow$ enumerate all k-cliques that contains $o^{(i)}$
3 add entry $(i, (NULL, INF))$ into the map G_b
4 **foreach** *clique* $g \in G^{(i)}$ **do**
5 $L \rightarrow$ fastest feasible route of g
6 **if** L *doesn't exist* **then continue**
7 $t_e \rightarrow$ the average extra time of reqeusts in L
8 **foreach** $o^{(j)} \in g$ **do**
9 $t_e^{(j)} \rightarrow$ the current optimal average extra time stored in $G_b[j]$
10 **if** $t_e < t_e^{(j)}$ **then**
11 replace $G_b[j]$ with $(j, (g, t_e))$
12 **return** *updated graph* $\mathcal{G}$*, updated map* G_b

B. Order Departure and Expiration Algorithm

In addition to new orders, updates to the graph can also be triggered by dispatch or rejection of orders, as well as expiration of groups or edges. We can process these updates in one subroutine, as shown in Algorithm 5. If a leave order is included in the current best groups of other orders, then the current best group of these orders needs to be recomputed. We use set C to store candidate orders to reduce duplicate calculations (lines 1-6). As for edge and group expiration, the process is similar to departure. We can use the group with a size of two to present the edge. Then, we only need to consider orders whose best group contains the orders in the group (lines 7-8). For each candidate order, we reuse and make a little Algorithm 4 with a small modification to generate the current best group (lines 9-10). The only different of the modification is there's no need to update the graph structure of the candidate orders.

C. Detailed Training Algorithm of DQN

Given historical data, we simulate the dispatch process using strategy Equation 5 in the order pool and collect experience transitions $(s_t, a, s_{t+\Delta t}, r_t)$. Algorithm 7 presents the details of an episode for experience collection and network update. The approach is implemented using the aforementioned framework, and we present the key sub-procedures for simplicity. We maintain a replay memory M to store the transitions. Additionally, due to the uncertainty of the next state for wait actions, we use a buffer B to temporarily store these transitions (line 3). To avoid the sparsity of the demand distribution, we use the first m orders to initialize it (line 4). Note that the update of the distribution only needs constant time (Line 7). Then, we decide the action for each order in the graph using the main network V and strategy Algorithm 2 (line 17-19). For the trade-off of exploration and exploitation, the action may be altered with probability ϵ (Line 20). If the order has reached the maximum waiting time, we also dispatch it (line 21). There are two kinds of situations that lead to the termination state: (1) timeout but not included in any group (line 11-15); (2) dispatched with a certain group (line 21-25). In these situations, the temporal transitions in buffer B will be popped and flushed with new transitions (Algorithm 6), while wait action leads to transitions temporarily stored in buffer B.

Algorithm 5: Order Departure or Group Expiration

Input: order group g, original graph $\mathcal{G}$, map G_b
Output: updated graph $\mathcal{G}$, updated map G_b

1 initialize order set C that need re-compute best group
2 **if** *departure* **then**
3 **foreach** $o^{(i)} \in g$ **do**
4 remove all edges related to order $o^{(i)} \in g$
5 $C^{(i)} \rightarrow$ the orders that have $o^{(i)}$ as a member of their best group
6 $C \rightarrow C \cup C^{(i)}$
7 **if** *expiration* **then**
8 $C \rightarrow$ the orders that have all orders in g as members of their best group
9 **foreach** $o^{(j)} \in C$ **do**
10 Order_Arrival($o^{(j)}$)
11 **return** *updated graph* $\mathcal{G}$*, updated map* G_b

Algorithm 6: Replace Terminate

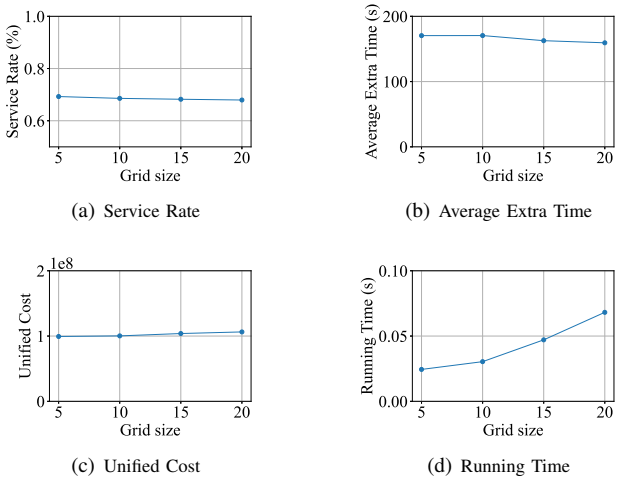
Input: original state s, action a, reward r, buffer B and memory M
Output: updated buffer B and memory M

1 $\delta' \leftarrow (s, a, o, TERMINATE)$
2 retrieve last transition $\delta = (s, a, o, s')$ from B
3 replace $s' \in \delta$ with s
4 flush δ, δ' into M
5 **return** *updated buffer* B *and memory* M

Algorithm 7: Deep-Q-Network (DQN) Leanring for Estimating Value Function

Input: A set W of m workers, a set O of n orders ordered by arriving time
Output: Learned state value function V

initialize shareability graph $\mathcal{G}$
initialize network V, $\hat{V}$
initialize replay memory M and buffer B
initialize the demand and supply distribution O, E, D by the first m orders
foreach $o^{(i)} \in O$ **do**
  insert $o^{(i)}$ into $\mathcal{G}$ and set $t^{(i)} \rightarrow t_s$
  update demand distribution O, E
  foreach $o^{(j)} \in \mathcal{G}$ **do**
    $g \leftarrow$ find a best group in $\mathcal{G}$ contains $o^{(j)}$
    if $g =$*NULL* **then**
      if *is_timeout*($o^{(j)}$) **then**
        remove $o^{(j)}$ from $\mathcal{G}$
        construct current state s
        Replace_Terminate(s, 0, 0, B, M)
        update O, E, D
      continue
    construct current states S of orders in g
    compute the state values V by S
    $a \leftarrow$ threshold_based_make_decision(g, V, t)
    alter a with another action with probability ϵ
    if $a = 1$ or *is_timeout*($o^{(j)}$) **then**
      foreach $o^{(k)} \in g$ **do**
        terminate($S[k]$, a, $p^{(k)} - t_d^{(k)}$, B, M)
      assign the g to a worker to serve.
      update O, E, D
    else
      $\delta' \leftarrow (S[i], a, \Delta t, NULL)$
      add δ' into B
return M



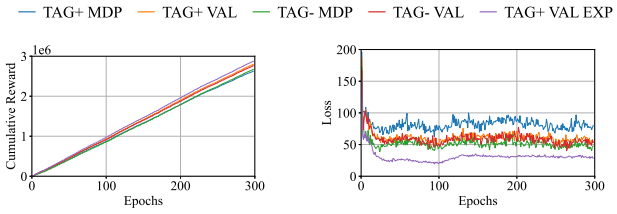
(a) Service Rate (b) Average Extra Time
(c) Unified Cost (d) Running Time

Fig. 8: Performance of varing grid size.

D. Impact of Varying Grid Size

To investigate the impact of grid size on algorithm performance, we conducted tests using the WATTER-expect algorithm on the NYC dataset. We divided the entire road network into grids of varying sizes: 5×5, 10×10, 15×15, and 20×20, while keeping the other parameters at their default values. The grid size primarily affects the accuracy of the model's predictions. More grids require more detailed training in order to achieve similar levels of accuracy. As the model takes distribution information into account, the number of grids is related to the feature dimensions, and using more grids results in a significant increase in algorithm running time. Figure 8 presents the results of varying the number of grids. After increasing the number of grids, there is little change in the unified cost, average extra time, and service rate. However, the average running time per request significantly increases.

E. Result of the Convergence Curves



(a) Cumulative Reward Trend (b) Loss Trend

Fig. 9: Convergence Curves.

Regarding the experiments on model convergence, we have prepared several training configurations. The labels TAG+ and TAG- indicate whether the target loss is used during training. The MDP configuration utilizes Q-learning with the value function for decision-making. In other words, the agent selects the action (dispatch or wait) that leads to the next state with higher reward as the final action. The VAL configuration employs the value function as the expected threshold for decision-making, similar to the approach described in Section V. Finally, the EXP label indicates that, in the first part of epochs, the GMM from Section V is used to generate expected thresholds, which are employed for decision-making to produce training experiences. In the later part of epochs, the value function is used as the expected threshold. In this section, there are a total of five training configurations: (1)TAG+ MDP, (2)TAG+ VAL, (3)TAG- MDP, (4)TAG- VAL, and (5)TAG+ VAL EXP. The last configuration TAG+ VAL EXP is used in WATTER-expect.

We conducted 10 repetitions of training on the NYC dataset, using default parameter configurations for each model. The resulting TD loss and reward mean values are reported to demonstrate the convergence, as depicted in Figure 9. Notably, the green curve, representing the pure Q-learning approach, exhibited the lowest cumulative reward. Conversely, the purple curve, representing the training approach used in WATTER-expect, achieved the highest cumulative reward. We observed that using the TAG+ label configuration resulted in higher

loss compared to the TAG- label configuration. This is due to partial fitting of the target value, resulting in a higher mean of the value function. For the VAL approach, when utilizing the GMM to generate training experiences in the early epochs (indicated by the purple curve), the convergence effect was smoother and yielded the highest cumulative reward. However, after the 100th epoch, the loss of the purple curve showed a slight increase. This occurred because before the 100th epoch, we used the GMM to generate training experiences, while afterward, we switched to using the value function for generating training experiences.