Unified Route Planning for Shared Mobility: An Insertion-based Framework

YONGXIN TONG, SKLSDE Lab and IRI, Beihang University, China
YUXIANG ZENG, The Hong Kong University of Science and Technology, China
ZIMU ZHOU, Singapore Management University, Singapore
LEI CHEN, The Hong Kong University of Science and Technology, China
KE XU, SKLSDE Lab and IRI, Beihang University, China

There has been a dramatic growth of shared mobility applications such as ride-sharing, food delivery, and crowdsourced parcel delivery. Shared mobility refers to transportation services that are shared among users, where a central issue is *route planning*. Given a set of workers and requests, route planning finds for each worker a route, i.e., a sequence of locations to pick up and drop off passengers/parcels that arrive from time to time, with different optimization objectives. Previous studies lack practicability due to their conflicted objectives and inefficiency in inserting a new request into a route, a basic operation called *insertion*. In addition, previous route planning solutions fail to exploit the appearance patterns of future requests hidden in historical data for optimization. In this paper, we present a unified formulation of route planning called URPSM. It has a well-defined parameterized objective function which eliminates the contradicted objectives in previous studies and enables flexible multi-objective route planning for shared mobility. We propose two insertion-based frameworks to solve the URPSM problem. The first is built upon the *plain-insertion* widely used in prior studies, which processes online requests only, whereas the second relies on a new insertion operator called *prophet-insertion* that handles both online and predicted requests. Novel dynamic programming algorithms are designed to accelerate both insertions to only linear time. Theoretical analysis shows that no online algorithm can have a constant competitive ratio for the URPSM problem under the competitive analysis model, yet our prophet-insertion-based framework can achieve a constant optimality ratio under the instance-optimality model. Extensive experimental results on real datasets show that our insertion-based solutions outperform the state-of-the-art algorithms in both effectiveness and efficiency by a large margin (e.g., up to 30× more effective in the objective and up to 20× faster).

Yongxin Tong and Ke Xu's work is partially supported by the National Key Research and Development Program of China under Grant No. 2018AAA0101100, the National Science Foundation of China (NSFC) under Grant Nos. 61822201, U1811463 and 62076017, the CCF-Huawei Database System Innovation Research Plan No. CCF-HuaweiDBIR2020008B, and the State Key Laboratory of Software Development Environment Open Funding No. SKLSDE-2020ZX-07. Yuxiang Zeng and Lei Chen's work is partially supported by National Key Research and Development Program of China Grant No. 2018AAA0101100, the Hong Kong RGC GRF Project 16209519, RIF Project R6020-19, CRF Project C6030-18G, C1031-18G, C5026-18G, AOE Project AoE/E-603/18, Theme-based project TRS T41-603/20R, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Microsoft Research Asia Collaborative Research Grant, HKUST-NAVER/LINE AI Lab, Didi-HKUST joint research lab, HKUST-Webank joint research lab grants.

Authors' addresses: Y. Tong and K. Xu, SKLSDE Lab and IRI, Beihang University, Xueyuan Road, Beijing, China, 100191; emails: {yxtong, kexu}@buaa.edu.cn; Y. Zeng and L. Chen, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong SAR, China; emails: {yzengal, leichen}@cse.ust.hk; Z. Zhou, Singapore Management University, 81 Victoria Street, Singapore, Singapore, 188065; email: zimuzhou@smu.edu.sg.

0362-5915/2022/05-ART2 $15.00
https://doi.org/10.1145/3488723

CCS Concepts: • **Information systems** → **Spatial-temporal systems**; **Geographic information systems;**

Additional Key Words and Phrases: Route planning, Ride-sharing, Insertion, Dynamic programming

ACM Reference format:
Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, and Ke Xu. 2022. Unified Route Planning for Shared Mobility: An Insertion-based Framework. *ACM Trans. Database Syst.* 47, 1, Article 2 (May 2022), 48 pages.
https://doi.org/10.1145/3488723

1 INTRODUCTION

Shared mobility refers to transportation services that are shared among users, such as ride-sharing, food delivery, and crowdsourced parcel delivery [45]. By altering routes and filling under-used vehicles, shared mobility mitigates pollution, reduces transportation costs, and provides last-mile delivery [53]. It is predicted as an efficient and sustainable alternative to urban transportation.

A key enabler for practical shared mobility is *route planning* among *workers* and *requests*. A worker can be a driver in ride-sharing services or a courier in food and parcel delivery services; and a request specifies an *origin* for pickup, and a *destination* for drop off. Route planning finds for each worker a *route* i.e., a sequence of locations to pick up and drop off passengers/parcels that arrive dynamically, with different optimization objectives.

Route planning for shared mobility has attracted extensive research interests from the database, data mining, and transportation science communities. Most studies consider a single or a subset of the following objectives: *(i)* minimizing the total travel distance [23, 25, 33, 39, 44, 47]; *(ii)* maximizing the number of served requests [12, 16, 23, 27, 44, 63]; and *(iii)* maximizing the total revenue [2, 3, 70, 71]. Many solutions are heuristic and rely on an operation called *insertion*, which inserts the origin and the destination of a new request into the current route [10, 12, 25, 33, 39, 44, 47, 63]. In practice, previous studies have the following limitations.

- *Limitation 1.* Existing proposals sometimes adopt multiple vague or even conflicted optimization objectives. For example, in [23, 25, 33, 39, 44], the goal is to minimize the total travel distance of workers without specifying how many requests should be served. Hence, an "optimal" solution is to serve no request at all, which contradicts common sense and the goal to maximize the number of served requests. A unified route planning problem with *flexible* and *consistent* optimization objectives is desirable for real-world applications.
- *Limitation 2.* The *insertion* operations in existing solutions [10, 25, 33, 34, 63] are inefficient for large-scale shared mobility platforms. It takes at least quadratic time complexity to insert a new request into a route, making insertion a bottleneck to process large numbers of requests.
- *Limitation 3.* Most studies on route planning (e.g., [2, 10, 25, 32–34]) assume a pure *online* setting [5], i.e., without any prior knowledge on the appearance of requests. Consequently, they fail to exploit the availability of big urban data and the advances in urban mobility prediction [20, 30, 48, 62], which may improve the effectiveness of route planning algorithms.

To address these limitations, we first define a new problem called **Unified Route Planning for Shared Mobility (URPSM)**. It unifies mainstream optimization objectives into a well-defined objective function where individual objectives are *compatibly* integrated. The URPSM problem also offers the flexibility to adjust the optimization goals for specific applications. We show that the three optimization goals above can be reduced as special cases of the URPSM problem.

As the efficiency bottleneck of many route planning algorithms is the *insertion* operation (*plain-insertion* in our context), we design a novel **dynamic programming (DP)** algorithm that reduces its time complexity from cubic or quadratic [10, 12, 25, 33, 63] to linear. The key insight is that dynamic programming can be utilized to find the best pickup location in $O(1)$ time for each enumerated drop-off location. Then, we devise a greedy-based framework using the DP-based plain-insertion to solve the URPSM problem under a pure *online* setting.

To further improve the effectiveness, we take advantage of the (offline) prediction on future requests. Specifically, we define a new insertion operator called *prophet-insertion*. Compared with the *plain-insertion* used in existing studies [25, 33, 54, 58], prophet-insertion plans the route for not only newly appeared requests, but also predicted requests. The routes of the predicted requests (a.k.a., guidance routes) are viewed as the guidance of the idle workers. That is, they will no longer move aimlessly [25] or passively stay in place [1, 33, 54, 58], but are guided to where new requests may appear in advance. Accordingly, we extend our DP-based techniques to achieve linear time complexity for prophet-insertion, as well as our greedy-based framework.

Last but not the least, unlike previous efforts that ignore the hardness of approximation analysis, we conduct a systematic theoretical analysis of the URPSM problem. We clarify and prove that there is no algorithm, either deterministic or randomized, with a constant competitive ratio for the URPSM problem and its special cases. The negative results imply that the *competitive analysis model* [5] for general online algorithms is unfit for the theoretical analysis of the solutions to the URPSM problem. Therefore, we take the *instance-optimality model* [17] to show the theoretical guarantees of our proposed algorithms.

Our main contributions are summarized as follows.

- We abstract a unified formulation of the route planning problem for shared mobility, i.e., URPSM, by a well-defined parameterized objective function. It eliminates the contradicted objectives in previous studies and benefits flexible multi-objective route planning in real-world shared mobility applications.
- We design a novel **dynamic programming (DP)** algorithm to accelerate the widely used *plain-insertion* operation. Our algorithm reduces the time complexity of this basic operation from cubic or quadratic to linear. On basis of our DP-based insertion, we further devise an effective and efficient framework called pGDP to solve the URPSM problem.
- We propose a new insertion operator called *prophet-insertion* exploiting predictions on future requests from historical data. We extend our DP-based techniques to achieve linear time complexity for prophet-insertion and propose a prediction-based solution named Prophet.
- We comprehensively analyze the hardness of approximation of the URPSM problem and *theoretical guarantees of our proposed algorithms*. Specifically, under the *competitive analysis model* [5], we prove that there is no polynomial-time online algorithm with a constant competitive ratio for the URPSM problem and its variants. However, under the *instance-optimality model* [17], our algorithm Prophet can achieve a constant optimality ratio (0.47) with high probability to maximize the number of served requests or the total revenue.
- Extensive experiments on real datasets with large-scale road networks show the superior performance gains of our proposed algorithms over the state-of-the-arts [1, 25, 33, 54]. Specifically, compared with the non-prediction-based methods [1, 25, 33], our algorithm pGDP is always more effective (e.g., up to 412× more effective than [1, 33] in the objective) and up to 23× faster than [1, 25]. Compared with the prediction-based method [54], pGDP is comparatively effective, but much more efficient (e.g., up to 80× faster than [54]). Our prediction-based method, Prophet, is always the most effective and also faster than [54]. For instance, Prophet yields up to 13× higher served rate (i.e., the percentage of the served requests among all the requests) than [54] with up to 20× lower time cost.

In the rest of this paper, we review related work in Section 2, formulate the URPSM problem, and discuss its generalizability as well as its hardness in Section 3. We present the plain-insertion-based framework in Section 4 and propose our linear-time plain-insertion in Section 5. We then introduce our prophet-insertion-based framework in Section 6 and linear-time prophet-insertion in Section 7. Finally, we present the experimental evaluations in Section 8 and conclude in Section 9.

2 RELATED WORK

Research on route planning for shared mobility (RPSM) dates back to the dial-a-ride problem proposed in 1975 [57] and has been studied by the database, data mining, and transportation science communities. This section briefly reviews different variants of the RPSM problem and their solutions.

2.1 Variants of RPSM Problems

An important setting in RPSM problems is *static* or *dynamic*. In a static (offline) RPSM problem, information of workers and requests is known in advance. Conversely, in a dynamic (online) setting, workers or requests appear dynamically, and requests need to be served within a short time or even immediately. Dynamic RPSM problems are more aligned with real-world shared mobility applications [2, 3, 12, 25, 33, 39, 47, 63] and hence will be our main focus.

Mainstream objectives of RPSM problems include minimizing the total travel distance [4, 8, 22], maximizing the number of served requests [12, 27, 31, 32, 44, 63], and maximizing the total revenue [2, 3, 54, 70, 71]. The total travel distance is the distance traveled by workers to serve requests. A short travel distance indicates a low travel cost and little pollution [24, 40, 46]. A large number of served requests contribute to the total revenue [63]. A more common goal is to minimize the total travel distance while serving all the requests [25, 33, 39, 47]. Other studies focus on minimizing the completion time of the requests [18], minimizing the average waiting time of the requests [65], maximizing the social utilities between workers and requests [10, 29], ensuring the price fairness [19], or answering the skyline queries [6, 9]. Our aim is to analyze the relationship among mainstream objectives and integrate them into a compatible and flexible formulation.

2.2 Solutions to Dynamic RPSM Problems

Many solutions to the dynamic RPSM problems have been proposed [12, 25, 33, 39, 47, 51, 63], where a core operation, called *insertion*, is widely utilized. Ma et al. [33, 34] use the enumeration strategy to search for the best insertion location, which needs to satisfy the constraints of the inserted requests. With additional constraints on the number of requests, the feasible insertions can be further reduced but optimal ones may also be mistakenly removed [39, 42]. Parallelism also applies to speed up insertion [39]. Insertion is frequently used in solutions to large-scale dynamic RPSM problems. However, the insertion has quadratic or even cubic time complexity, which is a bottleneck of efficiency. This motivates us to devise a linear-time insertion algorithm.

Solutions to the dynamic RPSM problems can be categorized into *batch-based* and *real-time*.

Batch-based Solutions. In these methods, each newly appeared request may need to wait for a specific time interval (i.e., the batch size) until its response [1, 4, 66, 70, 71]). Specifically, Alonso-Mora et al. [1] first generate many groups of requests which may be shared in each batch, and then iteratively insert each group of requests into the current route of each worker. Zheng et al. [70, 71] and Bei et al. [4] apply similar ideas as in [1]. Although they all use a bipartite graph to represent the relationships between the groups of requests and the workers, they focus on different objectives. Zheng et al. [70, 71] and Zeng et al. [66] maximize the total revenue of the platform. Bei et al. [4] minimize the total travel distance when each request has no deadline and the capacity of any worker is 2. Generally speaking, the size of the batch largely determines the time efficiency

of the batch-based solutions. For example, in our experiments, the average response time of [1] is nearly half of the batch size (e.g., 60s). This is because the requests, which are released at the beginning of each duration of 60s, are processed together at the end of the duration.

Real-Time Solutions. In these methods, the decision on whether there is an available worker who can serve the request is made right upon the request [25, 31–33, 54]. For example, for each newly appeared request, Ma et al. [33, 34] first search a set of candidate workers through grid index and then insert the request to the candidate with minimum increased distance. Huang et al. [25] propose an index (kinetic tree) to store all possible routes and use a similar insertion procedure to minimize the total travel distance. Luo et al. [32] propose a novel index for the road network to calculate the lower bound of the shortest distances in order to improve time efficiency. Wang et al. [54] consider the balance between future demand and current supply when planning the routes. Liu et al. [31] study a special scenario, where some requests may be invisible to the platform.

Due to the vast literature on the solutions to the RPSM problem, we compare our methods with the most representative and competitive schemes as baselines.

Specifically, we analyze the results of [1] to show that batch-based solutions often hardly can achieve real-time efficiency. We empirically compare T-share [33], the first work in the database community to study dynamic ride-sharing. We also empirically compare Kinetic [25], since their designed data structures, *kinetic trees*, are often used in existing studies (e.g., [9, 47, 69]). Finally, we compare a recent work [54] which utilizes predictions to outperform other solutions (e.g., [64]).

We exclude [32] in our baselines, because their road network index is orthogonal to our proposed solutions (e.g., used as the distance lower bound in Section 5.2.4). We also exclude [31] since it only applies to situations where drivers can secretly pick passengers without informing platforms.

A preliminary version of this work has been published in [50]. The new contributions over the preliminary work [50] are highlighted as follows.

(1) We propose a new solution framework (Section 6) based on a new insertion operator called *prophet-insertion* (Section 7). The solution in the preliminary version relies on the *plain-insertion*, which handles online requests only. In comparison, the new framework is more effective since the prophet-insertion processes both *online* and *predicted* requests. Moreover, we devise non-trivial DP-based techniques to accelerate the prophet-insertion from cubic-time to linear-time (Section 7).

(2) We exploit the instance-optimality model, which is better suited to differentiate solutions to the URPSM problem than the competitive analysis model in the preliminary version (Section 3.3). Our prophet-insertion-based framework can achieve a constant optimality ratio in maximizing the number of served requests or total revenue, while the plain-insertion-based method in [50] cannot.

(3) We conduct experiments with more state-of-the-art algorithms on larger datasets. Evaluations show that our new framework is up to 30× more effective than the solution in [50] (Section 8).

3 PROBLEM STATEMENT

In this section, we introduce the basic concepts (Section 3.1) and define the URPSM problem (Section 3.2), which unifies the objective functions of many prior studies. We also present our analysis models and hardness results in Section 3.3. The major notations are listed in Table 2.

3.1 Basic Concepts

Definition 1 (Road Network). A road network is denoted by an undirected graph $G = (V, E)$ with a vertex set V and an edge set E. Each edge $(u, v) \in E$ is associated with a travel time $\mathrm{dis}(u, v)$.

Definition 2 (Worker). A worker $w = \langle o_w, c_w \rangle$ has an initial location o_w and a capacity c_w.

The capacity c_w is the maximum number of passengers a taxi can take or the maximum number of items a courier's box can contain at any time. $W = \{w_1, \ldots, w_{|W|}\}$ represents all the workers.

Definition 3 (Request). A request is denoted by $r = \langle o_r, d_r, t_r, e_r, p_r, c_r \rangle$ with an origin o_r, a destination d_r, and a size c_r. It appears on the shared mobility platform (platform for short) at the release time t_r and needs to be served before the deadline e_r. A request is **served** if *(1)* a worker first picks up r at o_r after t_r; and *(2)* the same worker then delivers r at d_r before e_r. If a request is not served (or rejected), the platform will receive a penalty p_r.

Here the size c_r specifies the number of passengers in a ride-sharing request or the number of items in a food-delivery order. Since it is difficult to serve every request in practice, a platform may reject a certain request, which incurs a loss, i.e., penalty p_r, due to the loss in income from the served requests or user experience. The penalty is application-specific. We use $R = \{r_1, \ldots, r_{|R|}\}$ to denote all the requests. We further denote R_w as the set of requests served by worker w, $R^+ = \bigcup_{w \in W} R_w$ as all the served requests, and $R^- = R - R^+$ as all the rejected requests.

Definition 4 (Route). A route of a worker w is denoted by $S_w = \langle l_0, l_1, \ldots, l_n \rangle$, where $l_0 = o_w$ is the worker's initial location, and $\langle l_1, \ldots, l_n \rangle$ is an ordered sequence of the origins and destinations of R_w, i.e., $l_i \in \{o_r \mid r \in R_w\} \cup \{d_r \mid r \in R_w\}$. A route is **feasible** if *(1)* $\forall r \in R_w$, o_r precedes d_r in the route S_w; *(2)* $\forall r \in R_w$, the time when w arrives at d_r is no later than the deadline e_r; *(3)* At any time, the total number of passengers/items (i.e., the total size of the requests) that have been picked up but not delivered, does not exceed the capacity c_w of the worker w.

We use $D(S_w)$ to denote the total travel time of S_w, i.e., $D(S_w) = \sum_{i=1}^{n} \text{dis}(l_{i-1}, l_i)$.

3.2 Unified Objective and URPSM Problem

Based on the basic concepts above, we first define a new problem called ***Unified Route Planning for Shared Mobility (URPSM)*** as follows.

Definition 5 (URPSM). Given a road network G, a set of workers W, a set of requests R which are only known at their released time, and a weight coefficient α, the URPSM problem is to find, for each worker $w \in W$, a route S_w, such that the unified cost $UC(W, R)$ is minimized

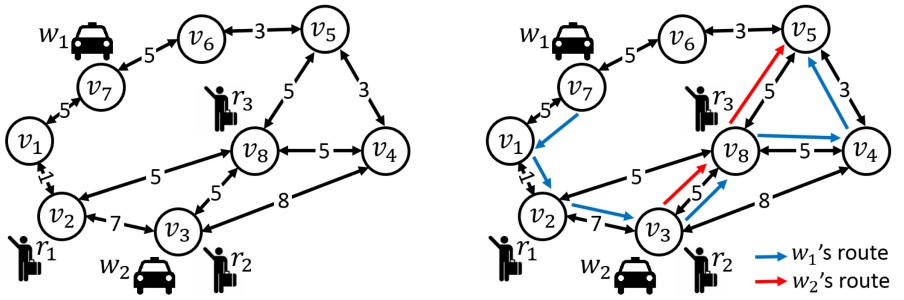
$$UC(W, R) = \alpha \sum_{w \in W} D(S_w) + \sum_{r \in R^-} p_r \tag{1}$$

and meets *(1)* the feasibility constraint: each worker is arranged a feasible route; *(2)* the invariable constraint: once requests are rejected, they cannot be revoked. Otherwise, they must be served.

We next illustrate the URPSM problem by a toy example.

Example 1. Suppose a ride-sharing platform with two workers (drivers) w_1-w_2 and three dynamically appeared requests r_1-r_3. The workers' initial locations are labeled on a road network with eight vertices v_1-v_8 as shown in Figure 1(a). Table 1 lists the details of the requests. We further assume each worker has a capacity of 4 (i.e., $c_w = 4$) and the weight coefficient $\alpha = 1$.

At time 5 (t_{r_1}), a request r_1 is released with an origin at v_2 and destination at v_4. To serve r_1, the platform needs to plan a route to pick up r_1 at v_2 and deliver it at v_4 before its deadline e_{r_1}. One feasible route for w_1 to serve r_1 is $\langle v_7(o_{w_1}), v_2, v_4 \rangle$. Between every two adjacent vertices in this route, the worker w_1 follows their shortest path and hence he can reach v_4 at time $5 + (5 + 1) + (5 + 5) = 21$. Specifically, w_1 starts from v_7 at time 5 and moves from v_7 to v_2 through v_1. Then, w_1 picks up r_1 at v_2 and travels from v_2 to v_4 through v_8. Finally, w_1 delivers r_1 at the destination v_4 before the deadline 28. The platform can also reject the request, which will incur a penalty $p_{r_1} = 20$. The URPSM problem plans routes for the workers and minimizes the unified cost, which is composed of both total travel cost and total penalty of the rejected requests. For instance, Figure 1(b) illustrates a feasible solution, where w_1 starts his route at time 0 and w_2 starts his route at time 10. Based on



(a) A road network, workers and requests (b) Planned routes S_{w_1} (blue) and S_{w_2} (red)

Fig. 1. The toy example of this paper (a road network, two workers w_1-w_2 and three requests r_1-r_3).

Table 1. Detailed Information of the Dynamically Appeared Requests r_1-r_3

Request	Release time t_r	Deadline e_r	Origin o_r	Destination d_r	Penalty p_r	Size c_r
r_1	5	28	v_2	v_4	20	1
r_2	10	31	v_3	v_5	10	1
r_3	11	26	v_8	v_5	9	1

Table 2. Summary of Major Notations

Notation	Description
R, W	a set of requests and a set of workers
o_r, d_r	the origin and destination of the request r
p_r, c_r	the penalty and size of the request r
o_w, c_w	the initial location and capacity of the worker w
R^+, R^-	a set of served requests and a set of unserved requests
$S_w, D(S_w)$	the route of worker w and its total travel time
α	the weight coefficient for unit travel time of workers
$UC(\cdot,\cdot)$	the unified cost (i.e., the objective of our URPSM problem)
$\text{dis}(\cdot,\cdot)$	the shortest travel time between two vertices on the road network

the planned routes, the total travel cost $\alpha \sum_{w \in W} D(S_w)$ is $1 \times (D(S_{w_1}) + D(S_{w_2})) = 26 + 10 = 36$. The total penalty of the rejected requests is 0, since all the requests can be served by the routes (e.g., r_1, r_3 are served by w_1 and r_2 is served by w_2). Thus, the unified cost is $36 + 0 = 36$.

Finally, we show many prior studies are special cases of URPSM with specific α and p_r settings.

- **Minimize the total travel time** [25, 33, 38, 41, 47]. When $\alpha = 1$ and $\forall r, p_r = \infty$, minimizing Equation (1) is equivalent to minimizing the total travel time while serving all the requests.
- **Maximize the number of served requests** [12, 16, 27, 63]. By setting $\alpha = 0$ and $\forall r, p_r = 1$, minimizing Equation (1) is equivalent to minimizing the number of unserved requests (i.e., maximizing the number of served requests) since every request's penalty is 1.
- **Maximize the total revenue** [2, 3, 70, 71]. The total revenue of the platform consists of the income of workers and the fare from the served requests. The income of a worker is related to the total working time and the income for unit time f_w. The fare of a request is relevant

to its length and the fare for unit time f_r. Then the total revenue is calculated as:

$$REV(W,R) = f_r \sum_{r \in R^+} \text{dis}(o_r, d_r) - f_w \sum_{w \in W} D(S_w) \tag{2}$$

Set $\alpha = f_w$ and $\forall r \in R, p_r = f_r \times \text{dis}(o_r, d_r)$:

$$UC(W,R) = f_w \sum_{w \in W} D(S_w) + f_r \sum_{r \in R^-} \text{dis}(o_r, d_r) \tag{3}$$

Substitute $R^+ = R - R^-$ and Equation (3) into Equation (2):

$$REV(W,R) = f_r \sum_{r \in R} \text{dis}(o_r, d_r) - UC(W,R) \tag{4}$$

Since the requests are given (i.e., $f_r \sum_{r \in R} \text{dis}(o_r, d_r)$ is a constant), minimizing $UC(W,R)$ in Equation (1) is equivalent to maximizing the total revenue $REV(W,R)$ in Equation (4).

Remark. Existing studies consider the requests' completion time [18] or waiting time [65] as the objectives. Here, a request's completion time is the time when it is delivered at the destination, and the waiting time is the duration from its release time to the completion time. Although they are not the special cases of our unified cost, the URPSM problem also considers them in the constraint. Specifically, each request r has a release time t_r and a deadline e_r in Definition 3. It restricts that r must be completed before e_r and its waiting time is no longer than $e_r - t_r$. To extend into their objectives, we can replace the total travel cost of workers (i.e., $\sum_{w \in W} D(S_w)$) in Equation (1) with the completion time of the last served request [18] or total waiting time of the served requests [65].

3.3 Analysis Models and Hardness Results

The URPSM problem is NP-hard since the above special cases are all NP-hard problems (e.g., [33]). To analyze the effectiveness guarantee, we present two analysis models, i.e., *competitive analysis model* (Section 3.3.1) and *instance-optimality model* (Section 3.3.2). Though competitive model [5] is one of the evaluation standards in theoretical computer science to analyze the effectiveness of an online algorithm, our hardness result shows that no algorithms can achieve a good theoretical guarantee (e.g., a constant competitive ratio). In other words, competitive analysis model fails to discriminate and to suggest good approaches to the URPSM problem. Thus, we introduce the instance-optimality model, which has been widely used in the database community [17].

3.3.1 Competitive Analysis Model and Hardness Result. Competitive ratio [5] is one of the most widely-used evaluation standards to analyze the effectiveness guarantee of an online algorithm. It measures how good an online algorithm is compared with the optimal result.

Definition 6 (Competitive Ratio). The competitive ratio of an online algorithm for the URPSM problem is the minimum ratio between the result of this online algorithm and the optimal result over all possible instances:

$$\text{competitive ratio} = \min_{\forall \text{ instance } (G,W,R)} \frac{UC(W,R)}{UC^*(W,R)} \tag{5}$$

where $UC(W,R)$ is the unified cost of this online algorithm and $UC^*(W,R)$ is the optimal result.

Though many existing papers have studied the special cases of URPSM, they lack theoretical analysis in terms of competitive ratio. In fact, only [2] proves that no deterministic algorithm can guarantee a constant competitive ratio to maximize the total revenue, but it is unknown whether

the conclusion applies to randomized algorithms. Thus, we further analyze the competitive hardness by studying whether any randomized algorithm can guarantee a constant competitive ratio in Theorem 1. If no such randomized algorithm exists, nor will any deterministic algorithm [5].

Theorem 1. *Neither a deterministic algorithm nor a randomized algorithm has a constant competitive ratio for the three special cases of the URPSM problem.*

The negative results for each special case are shown by Lemmas 1, 2, and 3.

Lemma 1. *When $\alpha = 0$ and $\forall r, p_r = 1$, no algorithm has a constant competitive ratio to maximize the number of served requests in the URPSM problem.*

Proof. We only need to show that no randomized algorithm can guarantee a constant competitive ratio. We first generate a distribution of the input and prove the expected value of any deterministic algorithm on this input is not constant (e.g., ∞). Then applying Yao's Principle [61], no randomized algorithm has a constant competitive ratio.

The distribution χ of the requests, workers, and road network is generated as follows: *(1)* We assume the road network G is an undirected cycle graph with $|V|$ vertices ($|V|$ is even) and the length of each edge is 1. *(2)* We assume a single worker with initial location $o_w = v_1$ and capacity $c_w = 2$. *(3)* A request r is released at time $t_r = |V|$ whose o_r is generated uniformly at random from all vertices V. We set $d_r = o_r$, $e_r = t_r + \epsilon$, $\epsilon > 0$ and $p_r = c_r = 1$.

Since the request is released at time $|V|$ and there are $|V|$ vertices in the graph, the worker in the optimal solution has enough time (i.e., $|V|$) to arrive at o_r when the request r is released. Hence, r can always be served by the optimal solution and the expected number of unserved requests is zero, i.e., $\mathrm{E}[UC^*(W, R)] = 0$.

Consider a generic deterministic online algorithm a which has its worker at point (not vertex) u when r is released. As long as the shortest distance between u and o_r is no greater than ϵ, it is able to serve r with a probability $\leq \frac{2\epsilon}{|V|}$. Since there is only one request, the expected number of unserved requests of algorithm a is $\mathrm{E}[UC(W, R)] \geq 1 - \frac{2\epsilon}{|V|}$. Hence, the competitive ratio is $(1 - 2\epsilon/|V|)/0$, which is unbounded. □

Lemma 2. *When $\alpha = f_w$ and $\forall r, p_r = f_r \times dis(o_r, d_r)$, no algorithm has a constant competitive ratio to maximize the total revenue in the URPSM problem.*

Proof. We prove Lemma 2 by adjusting the setting of the distribution in the proof of Lemma 1. Specifically, we generate the distribution d_r for the request r as follows. d_r is always chosen from a vertex in the cycle graph whose distance from o_r is $|V|/2$. Because the distance from the location of worker and o_r is no more than $|V|/2$ on an undirected cycle graph, and $\mathrm{dis}(o_r, d_r) = |V|/2$, the worker will move another $|V|/2$ to serve r. Therefore, the total travel distance of the worker is no more than $|V|/2 + |V|/2 = |V|$. We also assume a sufficiently large f_r, otherwise an optimal solution may reject r when the total distance of the worker is close to $|V|$. Then, we have

$$\mathrm{E}[UC^*(W, R)] \leq \alpha|V| = f_w|V| \text{ and } \mathrm{E}[UC(W, R)] \geq \left(1 - \frac{2\epsilon}{|V|}\right) \cdot p_r = \left(1 - \frac{2\epsilon}{|V|}\right) \cdot f_r \cdot \frac{|V|}{2}$$

If ϵ is small enough, then the competitive ratio becomes to $\Omega(\frac{f_r}{f_w})$, which is not constant. □

Lemma 3. *When $\alpha = 1$ and $\forall r, p_r = \infty$, no algorithm has a constant competitive ratio to minimize the total travel time in the URPSM problem.*

Proof. We prove Lemma 3 using the distribution in the proof of Lemma 1. According to previous analysis, the total distance of the optimal route under this distribution is bounded by $|V|$ and any

deterministic algorithm has probability of $1 - \frac{2\epsilon}{|V|}$ to reject r. Thus, we have

$$\mathrm{E}[UC^*(W,R)] \leq \alpha|V| = |V| \text{ and } \mathrm{E}[UC(W,R)] \geq \left(1 - \frac{2\epsilon}{|V|}\right) \cdot p_r$$

So the competitive ratio is worse than $\frac{p_r}{|V|}(1 - \frac{2\epsilon}{|V|})$. By setting a sufficiently small ϵ and $p_r = \infty$, the above competitive ratio becomes unbounded. □

Remark. There exists a special case of the URPSM problem, where a constant competitive ratio may exist. For example, when the penalty of every request is extremely small (e.g., $O(1/|R|)$), a simple solution, which rejects all requests, has a unified cost of $O(1)$. Thus, this solution may have a constant competitive ratio under this case. However, this case is meaningless in practice, since a real-world platform often aims to avoid rejecting the requests by setting large penalties. For the general URPSM problem, no algorithm can achieve a constant competitive ratio, since the competitive ratio in Definition 6 depends on the worst case (e.g., the special cases in Lemma 1–3).

3.3.2 Instance-Optimality Model. The negative result in Theorem 1 indicates that the competitive model fails to differentiate good solutions to the URPSM problem, as no algorithm can have a good guarantee under the model. Thus, we focus on another analysis model, *instance-optimality model*, in the rest of this paper. The instance-optimality model was first proposed in [17] and has been widely used in the database community. It measures how good a specific algorithm is over a family of algorithms on every instance of a dataset.

Definition 7 (Instance-Optimality). Given a family of online algorithms A and a dataset I with multiple instances, we say an online algorithm a is **instance optimal** if (1) $a \in A$ and (2) for every online algorithm $b \in A$ and every instance $(G, W, R) \in I$ we have

$$UC_a(W,R) \leq \rho \times UC_b(W,R) + o(1) \tag{6}$$

where $UC_a(W,R)$ and $UC_b(W,R)$ denote the unified cost of algorithm a and algorithm b, respectively, and ρ is also known as **optimality ratio**.

Compared with the competitive analysis model (Definition 6), the instance-optimality model is also *practical* and *powerful* due to the following reasons.

- It is more practical since it compares with the online algorithms. On the contrary, the competitive model compares with the optimal result, i.e., the result of an offline optimal algorithm that knows the workers and tasks beforehand.
- It is also powerful since instance-optimality corresponds to optimality in every instance.

We will introduce the scope of the instance-optimality model used in this paper in Section 4.3, including the algorithm family A and the dataset I in Definition 7.

4 PLAIN-INSERTION-BASED FRAMEWORK

Solutions built upon the *plain-insertion* are widely adopted for the variants of the URPSM problem. This section presents the plain-insertion-based framework and analyzes its instance-optimality. We review a prevalent plain-insertion in Section 5.1 and propose our linear-time plain-insertion in Section 5.2.

4.1 Framework Overview

Existing studies [25, 33, 54] solve the URPSM problem greedily. Specifically, for each newly appeared request, if it is not rejected, then it is assigned to the worker who can serve the request

with the minimum increased travel time. An *insertion* operation is used to calculate the minimum increased travel time. Formally, an insertion operation can be defined as follows [25, 26, 33].

Definition 8 (Plain-Insertion). Given a worker w with his current route S_w containing $n+1$ locations $(l_0, \ldots, l_n)$, and a new request r, the plain-insertion finds a new feasible route S_w^* with the minimum increased travel time to serve r by inserting both o_r and d_r into S_w, such that the order of locations in S_w remains the same in S_w^*.

By inserting the new request that has a minimum increased travel time, it also minimizes the total travel time. Thus, the goal is aligned with our URPSM problem, which minimizes the sum of weighted total travel time and the total penalty of unserved requests. The plain-insertion operation also keeps the relative order of the n locations unchanged. This improves efficiency by avoiding enumerating all possible routes (e.g., the number of these routes is $(n+2)!/2^{(0.5n+1)}$ [40]).

4.2 Framework Details

Algorithm 1 shows the plain-insertion-based framework. In line 1, we construct the grid index and initialize R^-. For each newly appeared request, we first determine a set *Cand* of candidate workers by a range filtering on the grid index with radius $e_r - t_r - \mathrm{dis}(o_r, d_r)$ (line 3). Lines 4–7 find a candidate worker among *Cand*, whose current route has the minimum increased travel time to insert the new request. Specifically, let w^* denote such a worker and Δ^* denote the minimum increased travel time (line 4). For each candidate worker, we calculate his increased travel time Δ to insert the new request (lines 5–6) and maintain the best one w^* with minimum increased travel time Δ^* (line 7). Lines 8–10 decide whether it accepts the new request or not based on its penalty and the increased travel cost (i.e., $\alpha \cdot \Delta^*$). If the penalty is larger than the increased travel cost, we decide to serve the request and update the route of w^* by plain-insertion (line 9). Otherwise, we can choose to reject the request (line 10).

Example 2. Back to our toy example. When the request r_1 is released at time 5, we assume that the candidate workers *Cand* are $\{w_1, w_2\}$ in line 3. In line 6, we calculate the increased travel time Δ to insert the new request for each worker in *Cand*. Based on Definition 8, the increased travel time for w_1 and w_2 are 16 and 17, respectively. The algorithm details of plain-insertion will be elaborated in Sections 5.1 and 5.2. Accordingly, we have $w^* = w_1$ and $\Delta^* = 16$ in line 8. Since the penalty of r_1 (20) is larger than the increased travel cost ($\alpha \cdot \Delta^* = 1 \times 16 = 16$), we decide to serve the request r_1 and assign it to the worker w_1. We also plan a route for w_1, i.e., $S_{w_1} = \langle v_7(o_{w_1}), v_2(o_{r_1}), v_4(d_{r_1})\rangle$. Between every two adjacent vertices in S_{w_1}, the worker w_1 can follow their shortest path. For instance, when w_1 travels from v_7 to v_2, he will first start from v_7, then come to v_1 and finally arrive at v_2. When the next request r_2 appears at time 10, w_1 moves to v_1 and w_2 stays at v_3. Similarly, in lines 5–6, we can calculate that $w^* = w_1$ has the minimum increased travel time (i.e., $\Delta^* = 8$), which will be further discussed in Example 3. As the penalty p_{r_2} is larger than the increased travel cost, we also decide to serve the request. We will assign r_2 to w_1 and update his route ($S_{w_1} = \langle v_7, v_2, v_3, v_4, v_5\rangle$). When the last request r_3 appears at time 11, it will be assigned to w_2 and the planned route for him is $S_{w_2} = \langle v_3, v_8, v_5\rangle$. Finally, we calculate the unified cost as $1 \times ((6+7+8+3)+(5+5))+0 = 34$, which is lower than the result in Figure 1(b).

Complexity Analysis. Assume the plain-insertion algorithm takes $O(T)$ time. The time complexity of Algorithm 1 is $O(|R||W|T)$, since there are $O(|R|)$ iterations in line 2 and $O(|W|)$ iterations in line 5, and other lines take no longer than $O(T)$ time.

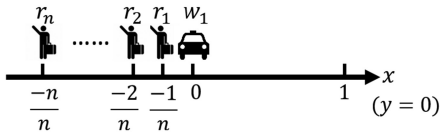


Fig. 2. Locations of the workers and requests in the worst case of Theorem 2.

ALGORITHM 1: Plain-insertion-based framework pruneGreedyDP (pGDP for short)

input : weight α, workers W and requests R
output: the unified cost $UC(W, R) \leftarrow \alpha \sum_{w \in W} D(S_w) + \sum_{r \in R^-} p_r$

```
Construct grid index and initialize R⁻ with ∅;
foreach newly appeared request r ∈ R do
    Cand ← filter the infeasible workers in W by grid index;
    w* ← not exists, Δ* ← ∞;
    foreach candidate worker w ∈ Cand do
        Δ ← plain-insertion(w, r);
        if Δ < Δ* then w* ← w, Δ* ← Δ;
    if w* is not empty and p_r ≥ α · Δ* then
        serve r and insert it into the route of w* by plain-insertion;
    else reject r and R⁻ ← R⁻ ∪ {r};
```

4.3 Instance-Optimality Analysis

We first introduce the **scope of our instance-optimality model** used in the rest of this paper. Based on Definition 7 (instance-optimality), we focus on the algorithm family of insertion-based online algorithms [52] and the datasets, where requests dynamically arrive following an **identical independent distribution (IID)** which is predictable. Specifically, this algorithm family uses the plain-insertion operator in Definition 8 to assign each request to the worker who has the minimum increased travel cost to serve it. We select this algorithm family due to two reasons: (1) the URPSM problem is an online problem and we have proved that there is no online algorithm with a constant competitive ratio. Thus, we can only consider heuristic algorithms instead of approximate solutions; (2) insertion-based online algorithms have been widely used to address the three mainstream cases of the URPSM problem in Section 3.2. Their performances have been verified by extensive experiments on real datasets (e.g., [25, 33, 34, 50, 58]). Besides, the assumption of IID is mild and commonly used in prior studies on online planning algorithms (e.g., [13–15, 36, 49, 59, 60, 68]), since many studies have been proposed to learn the arrival patterns of online requests (e.g., [20, 30, 48, 62, 67]).

Then, we prove the optimality ratio of Algorithm 1 under this instance-optimality model as follows.

Theorem 2. *The optimality ratio of Algorithm 1 is unbounded.*

Proof. To prove this result, we first create a worse case as shown in Figure 2. In this worst case, there is only one worker w_1 whose initial location is $(0, 0)$ and there are n requests $r_1, r_2, \ldots, r_n$. The requests' distribution is as follows. For each request r_i, its release time is $\frac{i}{n}$, its deadline $\frac{(i+1)}{n}$, and its origin and destination are $(\frac{-i}{n}, 0)$ and $(\frac{-(i+1)}{n}, 0)$, respectively. Besides, the weight α, the capacity or speed of the worker, and the penalty or size of each request are all 1. We assume that the shortest travel distance between two locations is the Euclidean distance of their coordinates.

To solve the above instance, the worker w_1 in Algorithm 1 will passively stay in place from time 0 to time $\frac{1}{n}$. When the request r_1 appears at time $\frac{1}{n}$, w_1 cannot serve this request earlier than its deadline $\frac{2}{n}$. Similarly, when any other request appears, w_1 cannot serve them either. As a result, the total travel time is 0, the total penalty is n, and hence the unified cost of Algorithm 1 is n.

Consider another algorithm in the family of insertion-based online solutions, which first makes the worker move from $(0, 0)$ to $(\frac{-1}{n}, 0)$ at the beginning and then starts to serve the requests by the same procedures in Algorithm 1. This time, when the request r_1 appears, the worker will arrive at $(\frac{-1}{n}, 0)$ and hence can serve it before the deadline. Similarly, when any other request r_i appears, the worker will locate at its origin and hence can serve it before the deadline. As a result, the total travel time is $\frac{(n+1)}{n}$, the total penalty is 0, and hence the unified cost is $1 + \frac{1}{n}$. Thus, the optimality ratio is $\Omega(\frac{n}{1+\frac{1}{n}}) = \Omega(n)$, which is as large as the number of requests (i.e., unbounded). □

Remark. The worst case in Theorem 2 may be counter examples for other methods (e.g., [54]), where idle workers will passively stay in place unless they are assigned to new requests. Although idle workers keep aimlessly moving in some other existing works (e.g., [25]), this will not help improve the optimality ratio. For instance, when there are massive places for this aimless worker w_1 to move, the probability for him to reach $(\frac{-1}{n}, 0)$ exactly at time $(\frac{1}{n}, 0)$ will be very low. Then, the (expected) optimality ratio will still be $\Omega(n)$. These results motivate us to propose a more flexible solution in Section 6, which exploits the prior knowledge on the appearance of requests by predictions.

5 ALGORITHMS FOR PLAIN-INSERTION

This section presents algorithms for the *plain-insertion* (i.e., handles online requests only). We review the quadratic-time plain-insertion in existing studies in Section 5.1 and propose our linear-time plain-insertion in Section 5.2. All the missing proofs involved in this section are in Appendix A.

5.1 Understanding Existing Algorithms for Plain-Insertion

Existing methods for the plain-insertion have cubic [33] or quadratic [25] time complexity. The **main idea** is to *(1)* enumerate all possible places ($O(n^2)$) for inserting o_r and d_r to obtain a new route S'_w; *(2)* check whether S'_w violates any constraint; and *(3)* replace S^*_w by S'_w if no constraint is violated and S'_w has a shorter increased travel time. The last two steps can be done in linear time and hence the straightforward method [33] takes *cubic time* complexity.

In this subsection, we present a simple *quadratic-time* insertion. Given a pair of places (denoted by (i, j)) for inserting o_r and d_r, a *quadratic-time* insertion takes $O(1)$ time to calculate the increased travel time denoted by $\Delta_{i,j}$ and checks the feasibility of the updated route after insertion.

5.1.1 Calculating Increased Travel Time $\Delta_{i,j}$ in $O(1)$ Time. Rather than calculate $\Delta_{i,j} = D(S'_w) - D(S_w)$ from scratch with $O(n)$ time, we calculate $\Delta_{i,j}$ in $O(1)$ time leveraging the concept of *detour*. Specifically, when inserting l_k between l_i and l_{i+1}, the detour $\det(l_i, l_k, l_{i+1})$ is defined as

$$\det(l_i, l_k, l_{i+1}) = \mathrm{dis}(l_i, l_k) + \mathrm{dis}(l_k, l_{i+1}) - \mathrm{dis}(l_i, l_{i+1}) \tag{7}$$

The *detour* is the increased travel time when l_k is inserted between l_i and l_{i+1}. Accordingly, $\Delta_{i,j}$ can be calculated in $O(1)$ time by Equation (8) using detours in Figure 3, where Figures 3(b) and 3(c) are two special cases (when $i = j$) while Figure 3(d) shows the general case (when $i < j$). Note that j can be equal to n in Figure 3(d) and hence l_{j+1} (i.e., l_{n+1}) becomes an empty location (denoted by $\emptyset$). For definition consistency, we assume $\mathrm{dis}(l_k, \emptyset) = \mathrm{dis}(\emptyset, l_k) = 0$ for any location l_k.

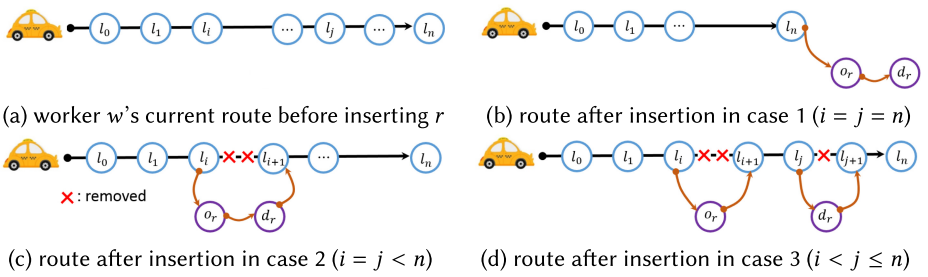


Fig. 3. Three cases of all $O(n^2)$ possible pairs (i, j).

$$\Delta_{i,j} = \begin{cases} \mathrm{dis}(l_n, o_r) + \mathrm{dis}(o_r, d_r), & \text{if } i = j = n \\ \mathrm{dis}(l_i, o_r) + \mathrm{dis}(o_r, d_r) + \mathrm{dis}(d_r, l_{i+1}) - \mathrm{dis}(l_i, l_{i+1}), & \text{if } i = j < n \\ \det(l_i, o_r, l_{i+1}) + \det(l_j, d_r, l_{j+1}), & \text{otherwise} \end{cases} \tag{8}$$

5.1.2 Checking Route Feasibility in $O(1)$ Time. To check whether a new route is feasible, we need to check all the three conditions in Definition 4. Specifically, condition (1) always holds since $i \leq j$ guarantees that o_r precedes d_r, where i and j denote the places to insert o_r and d_r, respectively. Since condition (2) is related to the deadlines of all the requests, we call it "deadline constraint" for short. As condition (3) is related to the capacity of the worker, it is called the "capacity constraint".

Checking Deadline Constraint. To check the deadline constraint in $O(1)$ time, we borrow the idea of slack time [26]. Denote $\mathrm{ddl}[k]$ as the latest time to arrive at l_k without violating the deadline constraint of the request at the location l_k. We set $\mathrm{ddl}[k]$ as $e_{r'} - \mathrm{dis}(o_{r'}, d_{r'})$ if l_k is the origin of a request r'; and $e_{r'}$ if l_k is the destination of a request r', i.e.,

$$\mathrm{ddl}[k] = \begin{cases} e_{r'} - \mathrm{dis}(o_{r'}, d_{r'}), & \text{if } l_k \text{ is the origin of a request } r' \\ e_{r'}, & \text{if } l_k \text{ is the destination of a request } r' \end{cases} \tag{9}$$

Denote $\mathrm{arr}[k]$ as the time when worker w arrives at the location l_k, i.e.,

$$\mathrm{arr}[k] = \mathrm{arr}[k-1] + \mathrm{dis}(l_{k-1}, l_k) \tag{10}$$

Further denote $\mathrm{slack}[k]$ as the maximal tolerable time for *detour* (i.e., slack time) between l_k and l_{k+1} to satisfy all the deadlines after l_k (excluded). Since a *detour* after l_k may cause the worker to violate the deadlines of $l_{k+1}, \ldots, l_n$, $\mathrm{slack}[k]$ should be small enough to satisfy the deadline of location l_{k+1} ($\mathrm{ddl}[k+1] - \mathrm{arr}[k+1]$), and all the deadlines after location l_{k+1} ($\mathrm{slack}[k+1]$). Hence, $\mathrm{slack}[k]$ is calculated as follows.

$$\mathrm{slack}[k] = \min\{\mathrm{slack}[k+1], \mathrm{ddl}[k+1] - \mathrm{arr}[k+1]\} \tag{11}$$

Finally, we check the deadline constraint by Lemma 4.

Lemma 4. *The deadline constraint is satisfied iff (1) $arr[i] + dis(l_i, o_r) \leq e_r$; (2) $det(l_i, o_r, l_{i+1}) \leq slack[i]$; (3) $arr[i] + dis(l_i, o_r) + dis(o_r, d_r) \leq e_r$ when $i = j$ (Case 1-2 in Figure 3(b)–3(c)) or $arr[j] + det(l_i, o_r, l_{i+1}) + dis(l_j, d_r) \leq e_r$ when $i < j$ (Case 3 in Figure 3(d)); and (4) $\Delta_{i,j} \leq slack[j]$.*

Checking Capacity Constraint. To check the capacity constraint in $O(1)$ time, we use $\mathrm{pick}[k]$ to denote the size of requests that are currently picked up yet not delivered. In practice, $\mathrm{pick}[k]$ represents the number of currently carried passengers/parcels in shared mobility. Thus, we have

$$\mathrm{pick}[k] = \begin{cases} \mathrm{pick}[k-1] + c_{r'}, & \text{if } l_k \text{ is the origin of a request } r' \\ \mathrm{pick}[k-1] - c_{r'}, & \text{if } l_k \text{ is the destination of a request } r' \end{cases} \tag{12}$$

ALGORITHM 2: Quadratic-time plain-insertion

input : a worker w with a route S_w and a request r
output: a new route S_w^* with the minimum increased travel time Δ after inserting r

1 $S_w^* \leftarrow S_w, \Delta \leftarrow \infty$;
2 Initialize ddl, arr, slack, pick by Equations (9) and (12);
3 **foreach** $i \leftarrow 0$ **to** n **do**
4 **if** *Lemma 4 (1) is violated* **then** break;
5 **if** *Lemma 4 (2) is violated* **then** continue;
6 **if** *Lemma 5 (1) is violated* **then** continue;
7 **foreach** $j \leftarrow i$ **to** n **do**
8 **if** *Lemma 5 (2) is violated* **then** break;
9 $\Delta_{i,j} \leftarrow$ calculate by Equation (8);
10 **if** *Lemma 4 (3)-(4) is violated* **then** break;
11 **if** $\Delta_{i,j} < \Delta$ **then** $\Delta, i^*, j^* \leftarrow \Delta_{i,j}, i, j$;
12 **if** $\Delta < \infty$ **then** $S_w^* \leftarrow$ insert o_r and d_r after locations l_{i^*} and l_{j^*}, respectively;

We then use Lemma 5 to check the capacity constraint.

Lemma 5. *The capacity constraint is satisfied iff (1)* $pick[i] \leq c_w - c_r$, *and (2)* $\forall k, i < k \leq j, pick[k] \leq c_w - c_r$.

5.1.3 Algorithm Details. Algorithm 2 shows the detailed procedure of the quadratic-time insertion. Specifically, Δ denotes the minimum increased travel time and S_w^* denotes the corresponding route after inserting the new request r. Line 2 performs the initialization by Equations (9) and (12). Line 3 iterates the possible place (i) to insert the new request's origin o_r and line 7 enumerates the possible place (j) to insert its destination d_r. To check the feasibility of the new route after insertion, Lines 4, 5, and 10 test the deadline constraint by Lemma 4, and lines 6 and 8 test the capacity constraint by Lemma 5. If the new route is feasible, we maintain the minimum increased travel time Δ and best places (i^*, j^*) for inserting o_r and d_r in line 11.

Complexity Analysis. Lines 2, 3, 7, and 12 take $O(n)$ time and the other lines take $O(1)$ time. Thus, the time complexity of Algorithm 2 is $O(n^2)$ and the total memory cost is $O(n)$. If the shortest distance query takes $O(q)$ time, the algorithm takes $O(n^2 q)$ time.

5.2 Our Linear-Time Plain-Insertion

In this subsection, we propose our linear-time plain-insertion method, which is based on dynamic programming. It finds the route with the minimum increased travel time without enumerating all possible pairs of places (i, j) for insertion. The linear-time plain-insertion is built upon the quadratic insertion algorithm, but leverages two insights: *(1)* it takes $O(n)$ time to find the best places (i^*, j^*) for the special cases (i.e., when $i = j$); and *(2)* for a given j, it takes $O(1)$ time to find the best i via dynamic programming in the general case (i.e., when $i < j$). The first insight is trivial because Algorithm 2 takes $O(1)$ time to check the feasibility and calculate increased travel time, i.e., it also takes $O(n)$ time when $i = j$. Hence, we mainly explain the second insight in the following.

5.2.1 Enumerating Delivery Locations Only. Instead of enumerating all possible pairs (i, j), our algorithm only enumerates the delivery location j to find the best route. Let Δ_j be the minimum

increased travel time among all possible insertion places such that d_r is inserted after l_j, i.e.,

$$\Delta_j = \min_{i<j} \Delta_{i,j} = \min_{i<j} \Big(\det(l_i, o_r, l_{i+1}) + \det(l_j, d_r, l_{j+1})\Big) = \det(l_j, d_r, l_{j+1}) + \min_{i<j} \det(l_i, o_r, l_{i+1}) \quad (13)$$

The first term ($\det(l_j, d_r, l_{j+1})$) in Equation (13) is the detour of inserting d_r after l_j, which is constant for a given j. The second term ($\min_{i<j} \det(l_i, o_r, l_{i+1})$) is the shortest detour of inserting o_r after l_i among all $i < j$. Thus, the main challenge is to calculate the second term in $O(1)$ time, i.e., finding the best pickup location (i.e., the best place for inserting o_r) in $O(1)$ time.

5.2.2 Finding the Best Pickup Location in $O(1)$ Time. We use $\text{dio}[j]$ to maintain the shortest detour for inserting o_r among all $i < j$ for a given j, i.e., $\text{dio}[j] = \min_{i<j} \det(l_i, o_r, l_{i+1})$. Accordingly, we can rewrite Equation (13) as:

$$\Delta_j = \det(l_j, d_r, l_{j+1}) + \text{dio}[j] \quad (14)$$

By *dynamic programming* (DP), $\text{dio}[j]$ can be efficiently pre-processed as:

$$\text{dio}[j] = \begin{cases} \infty, & \text{if } \text{pick}[j-1] > c_w - c_r \\ \text{dio}[j-1], & \text{if } \det(l_{j-1}, o_r, l_j) > \text{slack}[j-1] \\ \min\{\text{dio}[j-1], \det(l_{j-1}, o_r, l_j)\}, & \text{otherwise} \end{cases} \quad (15)$$

Correctness. We first prove the last case based on the definition of $\text{dio}[j]$ as follows.

$$\text{dio}[j] = \min\{\det(l_{j-1}, o_r, l_j), \min_{i<j-1} \det(l_i, o_r, l_{i+1})\} = \min\{\text{dio}[j-1], \det(l_{j-1}, o_r, l_j)\}$$

For the first two cases, we safely use $\text{dio}[j] = \infty$ to denote the situation that constraint is violated when inserting o_r at such a place. For instance, when $\text{pick}[j-1] > c_w - c_r$ (i.e., the first case), Lemma 5 will be violated. In the other case, when $\det(l_{j-1}, o_r, l_j) > \text{slack}[j-1]$, Lemma 4 (2) will be violated by inserting o_r after l_{j-1}. However, some place before l_{j-1} may be still feasible to insert o_r, so we set $\text{dio}[j]$ as $\text{dio}[j-1]$.

Let $\text{plc}[j]$ be the best place to insert o_r (corresponds to $\text{dio}[j]$). We can also update $\text{plc}[j]$ by DP.

$$\text{plc}[j] = \begin{cases} \text{not exists}, & \text{if } \text{pick}[j-1] > c_w - c_r \\ \text{plc}[j-1], & \text{if } \det(l_{j-1}, o_r, l_j) > \text{slack}[j-1] \\ \text{plc}[j-1], & \text{if } \text{dio}[j-1] < \det(l_{j-1}, o_r, l_j) \\ j-1, & \text{otherwise} \end{cases} \quad (16)$$

Correctness. If no constraints are violated when inserting at the places ($\text{plc}[j]$ and j), then we obviously obtain the best feasible route for the given j. However, when some constraint is violated when inserting o_r after $\text{plc}[j]$, it is unknown whether no other $i \neq \text{plc}[j]$ could form a feasible route. We prove the correctness by Lemma 6.

Lemma 6. *For a given j, if $plc[j]$ violates constraints, then other $i \neq plc[j]$ also violates constraints.*

Based on the proof of Lemma 6 and the definitions of $\text{dio}[\cdot]$ and $\text{plc}[\cdot]$, for the given j, we can simplify the feasibility tests in Lemma 4–5 by Corollary 1.

Corollary 1. *For the given j, a feasible i for inserting o_r exists iff (1) $pick[j] \leq c_w - c_r$, (2) $arr[j] + dio[j] + dis(l_j, d_r) \leq e_r$, and (3) $dio[j] + det(l_j, d_r, l_{j+1}) \leq slack[j]$.*

Table 3. ddl[·], arr[·], pick[·], slack[·], dio[·] and plc[·] in Example 3

k	ddl[k]	arr[k]	pick[k]	slack[k]	dio[k]	plc[k]
0	0	10	0	7	∞	doesn't exist
1	18	11	1	7	∞	doesn't exist
2	28	21	0	∞	5	1

ALGORITHM 3: Linear-time plain-insertion

input : a worker w with a route S_w and a request r
output: a new route S_w^* with the minimum increased travel time Δ after inserting r

1 $S_w^* \leftarrow S_w, \Delta \leftarrow \infty$;
2 Initialize ddl, arr, slack, pick by Equations (9)–(12) and dio, plc by Equations (15)–(16);
3 **foreach** $j \leftarrow 0$ **to** n **do**
4 Update $\Delta, i^*, j^* \leftarrow$ handle the special case when $i = j$ by Algorithm 2;
5 **if** $j > 0$ *and Corollary 1 is satisfiable* **then**
6 $\Delta_j \leftarrow \det(l_j, d_r, l_{j+1}) + \text{dio}[j]$ in Equation (14);
7 **if** $\Delta_j < \Delta$ **then** $\Delta, i^*, j^* \leftarrow \Delta_j, \text{plc}[j], j$;
8 **if** $\Delta < \infty$ **then** $S_w^* \leftarrow$ insert o_r and d_r after locations l_{i^*} and l_{j^*}, respectively;

5.2.3 Putting It Together. Algorithm 3 is our linear-time plain-insertion algorithm. In line 2, we perform the initialization by Equations (9)–(12) and Equations (15)–(16). We handle the special case (when $i = j$) using the same way as in Algorithm 2. In lines 5–7, we handle the general case (when $i < j$). Specifically, we first check the constraints by Corollary 1. If no constraint is violated, we calculate the minimum increased travel time Δ_j and the best pickup location plc[j] for the given j. In line 7, we maintain Δ and best places (i^*, j^*) for inserting o_r and d_r.

Example 3. Back to our toy example. Suppose w_1 is assigned to serve r_1 following the route $S_{w_1} = \langle v_7, v_2, v_4 \rangle$ (i.e., $n = 2$) when r_1 appears at time 5. When r_2 appears at time 10, w_1 moves to v_1. So there are another 2 vertices in S_{w_1} except for the current location of w_1 (i.e., v_1). If we insert r_2 into the current route S_{w_1}, the arrays in line 2 of Algorithm 3 are initialized as in Table 3. Specifically, ddl[0] = 0 since $l_0 = v_1$ is neither origin nor destination of r_1. arr[0] = 10 since current time is 10. pick[0] = 0 because w_1 has not picked up any request yet. For $k > 0$, ddl[k], arr[k], pick[k] are initialized using Equations (9), (10) and (12). For example, ddl[1] = $e_{r_1} - \text{dis}(o_{r_1}, d_{r_1}) = 28 - 10 = 18$, ddl[2] = $e_{r_1} = 28$, since $l_1 = v_2$ is o_{r_1} and $l_2 = v_4$ is d_{r_1}. arr[1] = arr[0] + $\text{dis}(v_1, v_2) = 10 + 1 = 11$, arr[2] = arr[1] + $\text{dis}(v_2, v_4) = 11 + 10 = 21$. According to Equation (11), $slack[k]$ is calculated from $k = n$ to 0. slack[n] is always initialized with ∞ because all requests would have been delivered after l_n. slack[1] = min{slack[2], ddl[2] − arr[2]} = 7, slack[0] = min{slack[1], ddl[1] − arr[1]} = 7. Besides, dio[0] = ∞, plc[0] = not exists, since a feasible i cannot exist for $j = 0$ in the general case as shown in Figure 3(d). By using Equations (15) and (16), we have dio[1] = dio[0], plc[1] = plc[0], since $\det(v_1, o_{r_2}, v_2) = \text{dis}(v_1, v_3) + \text{dis}(v_3, v_2) - \text{dis}(v_1, v_2) = 8 + 7 - 1 = 14 >$ slack[0]. We then update dio[2] = min{dio[1], $\det(v_2, o_{r_2}, v_4)$} = 5, because pick[1] = $1 \le c_w - c_r = 3$ and $\det(v_2, o_{r_2}, v_4) = 7 + 8 - 10 = 5 <$ slack[1] in Equation (15). Similarly, plc[2] = 2 − 1 = 1 since dio[1] > $\det(v_2, o_{r_2}, v_4)$ according to Equation (16). Then, lines 3–7 of Algorithm 3 work as follows. When $j = 0$, line 4 (i.e., route $\langle v_1, v_3(o_{r_2}), v_5(d_{r_2}), v_2, v_4 \rangle$ as Figure 3(c)) violates Lemma 4 (4), because $\Delta_{i,j} = \text{dis}(v_1, v_3) + \text{dis}(v_3, v_5) + \text{dis}(v_5, v_2) - \text{dis}(v_1, v_2) = 8+10+10-1 = 27 >$ slack[0]. When $j = 1$, line 4 (i.e., route $\langle v_1, v_2, v_3(o_{r_2}), v_5(d_{r_2}), v_4 \rangle$ as Figure 3(c)) also violates Lemma 4 (4), because $\Delta_{i,j} = \text{dis}(v_2, v_3) + \text{dis}(v_3, v_5) + \text{dis}(v_5, v_4) - \text{dis}(v_2, v_4) = 7 + 10 + 3 - 10 = 10 >$ slack[1]. In line 5, Corollary 1 (2) is violated because dio[1] = ∞. Finally,

when $j = 2$, line 4 (i.e., route $\langle v_1, v_2, v_4, v_3(o_{r_2}), v_5(d_{r_2})\rangle$ as Figure 3(b)) violates Lemma 4 (3), because $\text{arr}[2] + \text{dis}(v_4, v_3) + \text{dis}(v_3, v_5) = 21 + 8 + 10 = 39 > e_{r_2}$. In line 5, all conditions in Corollary 1 are satisfied. Thus, $\Delta_2 = \text{det}(v_4, d_{r_2}, \emptyset) + \text{dio}[2] = (3 + 0 - 0) + 5 = 8$ in line 6 and $\Delta = \Delta_2 = 8, i^* = \text{plc}[2] = 1, j^* = 2$ in line 7. In line 8, S_w^* becomes $\langle v_1, v_2, v_3, v_4, v_5\rangle$ by inserting $o_{r_2} = v_3$ after location $l_{i^*} = v_2$ and $d_{r_2} = v_5$ after location $l_{j^*} = v_4$.

Complexity Analysis. Lines 2–4 and 8 take $O(n)$ time and the other lines take $O(1)$ time. Thus, the time complexity of Algorithm 3 is $O(n)$ and space complexity is $O(n)$. If the shortest distance query takes $O(q)$ time, the algorithm takes $O(nq)$ time.

5.2.4 Optimization by Pruning. Despite numerous pruning strategies [2, 10, 25] to filter candidate workers to serve the new request, they mostly rely on the duration of deadlines and grid indices. They become ineffective with long deadlines of requests or large number of workers. Hence, we propose a new pruning strategy to filter workers leveraging the lower bound of the increased time (denoted by $\Delta^{\downarrow}$) when inserting a new request. The pruning strategy is based on Lemma 7.

Lemma 7. *Assume candidate workers Cand in line 3 of Algorithm 1 are already sorted in an ascending order by $\Delta^{\downarrow}$. If w_i is ahead of w_{i+1} in Cand and increased time Δ of w_i is smaller than the lower bound $\Delta^{\downarrow}$ of w_{i+1}, we can ignore all workers after w_i (i.e., break the iterations in lines 5–7 of Algorithm 1).*

Implementation Details. To calculate the lower bound of the increased travel time (i.e., $\Delta^{\downarrow}$), our idea is to replace the real travel time with the corresponding lower bound (denoted by $\text{dis}^{\downarrow}$) in Algorithm 3, e.g., the Euclidean distance between the vertices' coordinates. Specifically, we use $\text{len}[i+1] = \text{arr}[i+1] - \text{arr}[i]$ to replace $\text{dis}(l_i, l_{i+1})$ (i.e., the travel time between l_{i+1} and l_i) in the current route and $\text{dis}^{\downarrow}(l_i, o_r)/\text{dis}^{\downarrow}(l_i, d_r)$ to replace the travel time between l_i and the new request's origin o_r/destination d_r.

Since function $\text{dis}^{\downarrow}$ denotes the lower bound of the travel time function dis, we can calculate the lower bound of detour in Equation (7) (denoted by $\text{det}^{\downarrow}$) as

$$\text{det}^{\downarrow}(l_i, l_k, l_{i+1}) = \text{dis}^{\downarrow}(l_i, l_k) + \text{dis}^{\downarrow}(l_k, l_{i+1}) - \text{len}[i+1] \text{ (where } l_k \in \{o_r, d_r\}) \tag{17}$$

Similarly, we can infer the lower bound of $\Delta_{i,j}$ in Equation (8) (i.e., line 4 in Algorithm 3) and the lower bound of Δ_j (i.e., line 6 in Algorithm 3) by Lemma 8.

Lemma 8. *Let $\mathcal{L} = dis(o_r, d_r)$ be the request r's length. The lower bound $\Delta_{i,j}^{\downarrow}$ of $\Delta_{i,j}$ is calculated as*

$$\Delta_{i,j}^{\downarrow} = \begin{cases} dis^{\downarrow}(l_n, o_r) + \mathcal{L}, & \text{if } i = j = n \\ dis^{\downarrow}(l_i, o_r) + \mathcal{L} + dis^{\downarrow}(d_r, l_{i+1}) - len[i+1], & \text{if } i = j < n \end{cases} \tag{18}$$

The lower bound of Δ_j (denoted by $\Delta_j^{\downarrow}$) is calculated as

$$\Delta_j^{\downarrow} = det^{\downarrow}(l_j, d_r, l_{j+1}) + dio^{\downarrow}[j] \tag{19}$$

where the lower bound of $dio[\cdot]$ in Equation (15) (denoted by $dio^{\downarrow}[\cdot]$) is pre-processed as

$$dio^{\downarrow}[j] = \begin{cases} \infty, & \text{if } pick[j-1] > c_w - c_r \\ dio^{\downarrow}[j-1], & \text{if } det^{\downarrow}(l_{j-1}, o_r, l_j) > slack[j-1] \\ \min\{dio^{\downarrow}[j-1], det^{\downarrow}(l_{j-1}, o_r, l_j)\}, & \text{otherwise} \end{cases} \tag{20}$$

Remark. The calculation in Lemma 8 is efficient, since it needs only one shortest distance query on the road network, i.e., $\mathcal{L} = \text{dis}(o_r, d_r)$. To efficiently implement Lemma 7, we can use the radix sort which takes $O(|W|)$ time complexity and hence the time complexity of Algorithm 1 will not

increase. Moreover, although we are using the lower bound of travel time, workers may still violate capacity constraints or deadline constraints, who can be pruned before the sort.

6 PROPHET-INSERTION-BASED FRAMEWORK

Although the *efficiency* of the plain-insertion-based framework is improved by our linear-time plain-insertion algorithm, our instance-optimality analysis shows that the effectiveness of the framework can be bad, especially when there are limited numbers of workers. In response, we design a new framework with a constant optimality ratio. The framework is based on a new insertion operation called *prophet-insertion*, where a worker can wait at some location until a request appears. In this section, we introduce the new framework and analyze its optimality ratio. We present the naive cubic-time prophet-insertion in Section 7.1 and a linear-time prophet-insertion in Section 7.2.

6.1 Framework Overview

The **basic idea** of our new framework is to exploit the prediction of requests to guide the current route of workers. That is, if the platform is a prophet that can make accurate predictions on the incoming requests, then the workers can arrive at the requests' origins in advance and serve more requests. The new framework exploits the recent success in predicting taxi requests in ride-sharing [20, 30, 48, 62]. Our new framework is still insertion-based, but differs in the assumptions on the prior knowledge of the requests. Specifically, when the request distributions are known beforehand, a worker can arrive early at the requests' origins and wait based on the predictions in the following new framework. When the request distributions are unknown, the worker does not have to wait at the request's origin in the aforementioned framework. This difference leads to a new and complex insertion operation (called "*prophet-insertion*") which will be explained in Section 7.

6.2 Framework Details

Our prophet-insertion-based framework processes the *predicted* and *online* requests differently. The *prophet-insertion* mainly differs from the *plain-insertion* in the definition of *routes* (see below).

6.2.1 Processing Predicted Requests. Let $\widetilde{R}$ be a set of predicted requests. We first introduce the *guidance route*. It differs from the plain route (Definition 4) in terms of *waiting time*.

Definition 9 (Guidance Route). Given a set of requests $\widetilde{R_w} \subseteq \widetilde{R}$ assigned to the worker w, a guidance route of this worker w is denoted by $\widetilde{S_w} = \langle (l_0, \text{wait}[0]), (l_1, \text{wait}[1]), \ldots, (l_n, \text{wait}[n]) \rangle$, where $l_0 = o_w$ is the worker's initial location, $l_1, \ldots, l_n$ are the requests' origins or destinations (i.e., $l_i \in \{o_r \mid r \in \widetilde{R_w}\} \cup \{d_r \mid r \in \widetilde{R_w}\}$), and $\text{wait}[i]$ is the waiting time for this worker to stay at the corresponding location l_i. A guidance route is **feasible** if *(1)* $\forall r \in \widetilde{R_w}$, o_r precedes d_r in the route S_w; *(2)* $\forall r \in \widetilde{R_w}$, the time when w leaves from o_r is no earlier than the release time t_r, and the time when w arrives at d_r is no later than the deadline e_r. *(3)* At any time, the total number of passengers/items (i.e., the total size of the requests) that have been picked up but not delivered, does not exceed the capacity of the worker.

Note that the plain route defined in Definition 4 is a special case of guidance route, where all the waiting time is 0. Based on Definition 9, the total travel time of a guidance route is defined as the sum of total moving time and total waiting time, i.e., $D(\widetilde{S_w}) = (\sum_{i=1}^{n} \text{dis}(l_{i-1}, l_i)) + (\sum_{i=0}^{n} \text{wait}[i])$. We can now extend the concept of *plain-insertion* to *prophet-insertion* as follows.

Definition 10 (Prophet-Insertion). Given a worker w with his current guidance route S_w containing $n + 1$ locations $(l_0, \ldots, l_n)$, and a new request r, the prophet-insertion operation finds a new

ALGORITHM 4: Processing predicted requests

input : Workers W and a set of predicted requests $\widetilde{R}$
output: A guidance route $\widetilde{S_w}$ for each worker $w \in W$

1 A set of hypothetical workers $\widetilde{W} \leftarrow \emptyset$;
2 **foreach** *next predicted request* $r \in \widetilde{R}$ **do**
3 $Cand \leftarrow$ filter the infeasible workers in $\widetilde{W}$ by grid index;
4 $\widetilde{w^*} \leftarrow \arg\min_{\widetilde{w} \in \widetilde{W}} \text{prophet-insert}(\widetilde{w}, r)$, $\Delta^* \leftarrow \min_{\widetilde{w} \in \widetilde{W}} \text{prophet-insert}(\widetilde{w}, r)$;
5 **if** $\widetilde{w^*}$ *is not empty* **then** insert r into the guidance route of $\widetilde{w^*}$ by prophet-insert$(\widetilde{w^*}, r)$;
6 **else** $\widetilde{w} \leftarrow$ a hypothetical worker at the location o_r to serve r, $\widetilde{W} \leftarrow \widetilde{W} \cup \{\widetilde{w}\}$;
7 $Q \leftarrow$ a max-heap of $\widetilde{W}$ based on the total number/revenue of the served requests in the guidance route of $\widetilde{w} \in \widetilde{W}$;
8 **while** Q *is not empty and* W *is not empty* **do**
9 $\widetilde{w^*} \leftarrow$ pop the top element from Q, $w \leftarrow$ the worker in W who can serve all the requests in the guidance route of $\widetilde{w^*}$ with minimum increased travel time;
10 **if** w *is not empty* **then** plan the guidance route of w based on $\widetilde{w^*}$, $W \leftarrow W \setminus \{w\}$;
11 **else** remove the first request in the guidance route of $\widetilde{w^*}$, $Q \leftarrow Q \cup \{\widetilde{w^*}\}$;

and feasible guidance route S_w^* with the minimum increased travel time to serve r by inserting both o_r and d_r into S_w, such that the order of locations in S_w remains the same in S_w^*.

For brevity, we use guidance route and route, prophet-insertion and insertion interchangeably in the prophet-insertion-based framework.

Basic Idea. Given predicted requests $\widetilde{R}$, we plan a guidance route $\widetilde{S_w}$ for each worker as follows.

- Assume sufficient hypothetical workers (denoted by $\widetilde{W}$) with initial locations at any origins.
- Sequentially insert all the requests and assign each request to the worker whose increased travel time is minimum.
- Pick $|W|$ hypothetical workers to plan the guidance routes for real workers W, whose guidance routes have the largest number of served requests or the highest total revenue.

Algorithm Details. Algorithm 4 processes the predicted requests. For each request r in $\widetilde{R}$, we first determine a set $Cand$ of candidate workers by grid index in line 3. Then, we apply the prophet-insertion to find the worker $\widetilde{w^*} \in \widetilde{W}$, who has the minimum increased travel time (Δ^*) to insert the predicted request r (line 4). If such a worker exists, we insert r in his guidance route in line 5. Otherwise, we create a new hypothetical worker $\widetilde{w}$ to serve r in line 6, whose initial location is the origin of r. When all the predicted requests have been inserted, we assign the requests in the routes of hypothetical workers to the (real) workers to construct their guidance routes. Specifically, we use a max-heap Q to maintain the top hypothetical worker $\widetilde{w^*}$ with the largest number of served requests or the highest total revenue in lines 7–11. For each $\widetilde{w^*}$, we iteratively try each worker in W to serve all the requests by leaving from his origin at time 0 and then following the guidance route of $\widetilde{w^*}$. Accordingly, we can find the worker w who has the minimum increased travel time to serve the requests and plan the guidance route for him (line 10). If such worker w does not exist, it indicates no worker can reach the first origin in the guidance route of $\widetilde{w^*}$ on time. So we remove the firstly picked request in the guidance route of $\widetilde{w^*}$ and push it back into the heap Q (line 11).

Example 4. Back to our toy example. For simplicity, we assume the prediction is accurate, i.e., $\widetilde{R} = R = \{r_1, r_2, r_3\}$. For the first predicted request r_1, we create a hypothetical worker $\widetilde{w_1}$

ALGORITHM 5: Prophet-insertion-based framework named Prophet

input : weight α, workers W, requests R, and predicted requests $\widetilde{R}$
output: the unified cost $UC(W,R) \leftarrow \alpha \sum_{w \in W} D(S_w) + \sum_{r \in R^-} p_r$

1 guidance routes $\{\widetilde{S_w}\} \leftarrow$ process the predicted requests $\widetilde{R}$ by Algorithm 4;
2 set $c_r \leftarrow 0, e_r \leftarrow \infty$ for every predicted request $r \in \widetilde{R}$;
3 **foreach** *request* $r \in R$ **do**
4 $Cand \leftarrow$ filter the infeasible workers in W by grid index;
5 $w^* \leftarrow$ not exists, $\Delta^* \leftarrow \infty$;
6 **foreach** *candidate worker* $w \in Cand$ **do**
7 $\Delta \leftarrow$ prophet-insert(w, r);
8 **if** $\Delta < \Delta^*$ **then** $w^* \leftarrow w, \Delta^* \leftarrow \Delta$;
9 **if** w^* *is not empty and* $p_r \geq \alpha \cdot \Delta^*$ **then** insert r it into the guidance route of w^*;
10 **else** reject r and $R^- \leftarrow R^- \cup \{r\}$;
11 Let the workers without any assignments keep moving by the guidance routes;

who locates at the initial location v_2 (i.e., o_{r_1}) with capacity 4 in line 6, since $\widetilde{W}$ is currently empty. $\widetilde{w_1}$ can serve r_1 by the guidance route $\langle (v_2, 0), (v_2, 5), (v_4, 0) \rangle$. For example, $\widetilde{w_1}$ will wait at the location $l_1 = v_2$ for time 5 until r_1 appears. When the next predicted request r_2 is iterated in line 2, we use prophet-insertion to insert it into the guidance route of $\widetilde{w_1}$. The algorithm details will be elaborated in Section 7, and we can calculate that $\widetilde{w^*} = \widetilde{w_1}, \Delta^* = 8$ (line 4) and the new guidance route is $\langle (v_2, 0), (v_2, 5), (v_3, 0), (v_4, 0), (v_5, 0) \rangle$ (line 5). Similarly, when the last predicted request r_3 is iterated in line 2, we can calculate that $\widetilde{w^*} = \widetilde{w_1}, \Delta^* = 2$ and the new guidance route is $\langle (v_2, 0), (v_2, 5), (v_3, 0), (v_8, 0), (v_4, 0), (v_5, 0), (v_5, 0) \rangle$. In line 9, we pop $\widetilde{w_1}$ from the heap Q and check whether the (real) workers w_1-w_2 can serve the predicted requests (i.e., r_1-r_3) in the guidance route of $\widetilde{w_1}$. For instance, a feasible guidance route for w_1 is $\langle (v_7(o_{w_1}), 0), (v_2, 0), (v_3, 0), (v_8, 0), (v_4, 0), (v_5, 0), (v_5, 0) \rangle$. The guidance route for w_2 is empty.

Complexity Analysis. Assume the prophet-insertion algorithm takes $O(T)$ time and hence the iterations in lines 1–6 take $O(|R||W|T)$ time. Lines 7–11 take at most $O(|R| \log |R| + |R||W|T)$ time, because there are at most $O(|R|)$ iterations in line 8 and lines 9–11 take at most $O(\log |R| + |W|T)$ time. We can use prophet-insertion to check whether a worker can serve all the requests in a guidance route in line 9. Thus, the time complexity of Algorithm 4 is $O(|R||W|T + |R| \log |R|)$.

6.2.2 Processing Online Requests. Our prophet-insertion-based framework (named Prophet) handles each online request based on the following **main ideas**.

- Insert the request into the guidance route of each worker.
- Pick the worker with the minimum increased travel time to serve the requests.
- Let workers without assignments keep moving by their guidance routes, since the online requests may appear near to the guidance routes with high probability based on the prediction.

Algorithm Details. Algorithm 5 processes the online requests. Specifically, we pre-process the guidance routes $\{\widetilde{S_w}\}$ by Algorithm 4 in line 1. In line 2, we set the size of each predicted requests $\widetilde{R}$ to 0 and its deadline to ∞, since they are not the actual requests. Otherwise, some actual requests cannot be inserted, because the predicted requests have occupied some capacities of the workers. For each online request r (line 3), we first get a set of candidate workers denoted by *Cand* in line 4. Then, we apply the prophet-insertion operation to pick the worker w^* who has the minimum

increased travel time Δ^* to serve the request r (lines 5–8). Next, we assign the request to the worker w^* (line 9), if the penalty is larger than the cost (i.e., $p_r \geq \alpha \cdot \Delta^*$). Otherwise, we reject the request (line 10). The workers who have no assignments will keep moving by the guidance routes (line 11).

Example 5. The guidance routes are $\widetilde{S_{w_1}} = \langle (v_7, 0), (v_2, 0), (v_3, 0), (v_8, 0), (v_4, 0), (v_5, 0), (v_5, 0) \rangle$ and $\widetilde{S_{w_2}} = \langle \rangle$. When the first online request r_1 is released at time 5, w_1 moves to v_1 and w_2 stays at v_3. Assume the candidate workers are $\{w_1, w_2\}$ (line 4). In line 7, we can calculate the minimum increased travel time $\Delta = 0$ and 17 to insert the new request for w_1 and w_2, respectively, where the calculation details will be elaborated in Sections 7.1 and 7.2. Accordingly, we have $w^* = w_1$ and $\Delta^* = 0$ in line 9. Since the penalty of r_1 ($p_{r_1} = 20$) is larger than the increased travel cost ($\alpha \cdot \Delta^* = 0$), we decide to serve the request r_1 and assign it to w_1. We also update his current guidance route, i.e., $\langle (v_1, 0), (v_2(o_{r_1}), 0), (v_2, 0), (v_3, 0), (v_8, 0), (v_4(d_{r_1}), 0), (v_4, 0), (v_5, 0), (v_5, 0) \rangle$. Similarly, when the other requests r_2 and r_3 appear, they will also be assigned to w_1 with the minimum increased travel time $\Delta^* = 0$. Accordingly, the moving trajectory of w_1 is marked by blue arrows in Figure 1(b) and w_2 always stays at his initial location v_3. Thus, the unified cost is $1 \times ((6 + 7 + 5 + 5 + 3) + 0) + 0 = 26$.

Complexity Analysis. Assume the prophet-insertion algorithm takes $O(T)$ time. By Algorithm 4, lines 1–2 take $O(|R||W|T + |R| \log |R|)$ time, where $O(|R|) = O(|\widetilde{R}|)$ in practice. For the other lines, line 3 has $O(|R|)$ iterations, line 6 has $O(|W|)$ iterations, and line 7 takes $O(T)$ time. Thus, lines 3–11 take $O(|R||W|T)$ time. Overall, the time complexity of Algorithm 5 is $O(|R||W|T + |R| \log |R|)$.

Remark. In practice, the traffic may become congested if too many workers with a long waiting time are guided to the same area that potentially has many requests. To solve this issue, we can borrow the techniques from existing work [7, 55, 56] on traffic congestion management. Specifically, a platform can control the guidance by adjusting the predicted requests if the future traffic becomes congested [56]. For example, we can control the number of concurrent requests in each area such that the number of workers guided to this area is below a reasonable threshold. This is an external factor of the URPSM problem. Our experimental evaluations do not consider this issue due to these reasons: (1) modeling a traffic jam involves many real-world factors [7, 55, 56], e.g., statistics of all vehicles, information of all transportation services, weathers, road conditions and local regulations; (2) this issue may also affect the existing baselines (e.g., [25, 54]), since workers may be self-motivated to stay in the areas with more requests to potentially get more assignments; and (3) we want to follow the experimental settings in the existing baselines [1, 25, 33, 54].

6.3 Instance-Optimality Analysis

Basic Idea. We analyze the *optimality ratio* of our prophet-insertion-based framework to show that it is nearly optimal among all the insertion-based algorithms. Our basic idea is as follows.

(1) We assume that the requests dynamically arrive following an **identical independent distribution (IID)** which is predictable (i.e., known beforehand). This assumption is mild and commonly used in prior studies on online algorithms (e.g., [13–15, 36, 49, 59, 60, 68]). Moreover, many studies have been proposed to learn the arrival patterns of online requests (e.g., [20, 30, 48, 62, 67]).
(2) We first analyze the algorithm of processing predicted requests in Lemma 9. We will prove Algorithm 4 can be used to obtain the upper bound of the number of served requests or the total revenue by any insertion-based solution.
(3) We next analyze the algorithm of processing online requests in Theorem 3. Since the predicted requests may not appear in practice (i.e., prediction can be wrong), we will prove that the optimality ratio of Algorithm 5 is a constant, which is better than 0.47 in practice.

ALGORITHM 6: Cubic-time prophet-insertion

input : a worker w with a route S_w and a request r
output: a new route S_w^* with the minimum increased travel time Δ after inserting r

```
1 S_w^* ← S_w, Δ ← ∞;
2 foreach i ← 0 to n do
3     foreach j ← i to n do
4         S'_w ← insert o_r and d_r after locations l_i and l_j, respectively;
5         Δ_{i,j} ← D(S'_w) − D(S_w);
6         if S'_w is feasible then
7             if Δ_{i,j} < Δ then S_w^* ← S'_w, Δ ← Δ_{i,j};
```

The proofs of Lemma 9 and Theorem 3 are presented in Appendix B.

Lemma 9. *Under the assumption of known IID, Algorithm 4 can be used to obtain the upper bound of the number of served requests or the total revenue by any insertion-based online algorithm.*

By Lemma 9, we can infer that the optimality ratio is optimal when the prediction is completely accurate. Theorem 3 presents the theoretical result when the prediction is inaccurate.

Theorem 3. *Under the assumption of known IID, Algorithm 5 has a constant (expected) optimality ratio to maximize the number of served requests or total revenue, where the constant is better than 0.47.*

7 ALGORITHMS FOR PROPHET-INSERTION

In this section, we present the cubic-time prophet-insertion in Section 7.1 and propose our linear-time prophet-insertion in Section 7.2. All missing proofs involved in this section are in Appendix B.

7.1 Naive Cubic-Time Prophet-Insertion

Basic Idea. An intuitive method to implement the prophet-insertion is to *(1)* enumerate all possible pairs (e.g., (i, j)) of places for inserting o_r and d_r to obtain a new route S'_w; *(2)* check whether the new route S'_w violates any constraint and then calculate the increased travel time by $D(S_w) - D(S'_w)$; and *(3)* replace S_w^* by S'_w if no constraint is violated and S'_w increases a shorter travel time.

Algorithm Details. Algorithm 6 illustrates the naive prophet-insertion. In lines 2–3, we enumerate all possible pairs (i, j) of insertion places for the new request's origin and destination. Then, we have a new route S'_w in line 4 and calculate the increased travel time ($\Delta_{i,j}$) in line 5. If S'_w is feasible, we maintain the currently best route S_w^* and the minimum increased travel time Δ in lines 6–7.

Example 6. In Example 4, a predicted request r_2 is inserted into the route $\langle (v_2, 0), (v_2, 5), (v_4, 0)\rangle$. We can use Algorithm 6 to calculate the minimum increased travel time. For example, when $i = 1, j = 2$, we obtain a new route $\langle (v_2, 0), (v_2, 5), (v_3(o_{r_2}), 0), (v_4, 0), (v_5(d_{r_2}), 0)\rangle$ in line 4. In line 5, we calculate its increased travel time, i.e., $\Delta_{i,j} = 23 - 15 = 8$. In line 6, we check the feasibility of this new route based on Definition 9. Line 7 ensures that S_w^* has the minimum increased travel time Δ.

Complexity Analysis. The time complexity of Algorithm 6 is $O(n^3)$ and its space complexity is $O(n)$. If the shortest distance query takes $O(q)$ time, the algorithm will take $O(n^3 q)$ time.

7.2 Our Linear-time Prophet-Insertion

Our linear-time prophet-insertion algorithm extends the optimization techniques in Section 5.2 for the plain-insertion. We first introduce the main idea of the linear-time prophet-insertion in Section 7.2.1, then elaborate on how to handle the special case ($i = j$) in Section 7.2.2 and general case ($i < j$) in Section 7.2.3, and finally present the complete algorithm in Section 7.2.4.

7.2.1 Overview. The **main idea** of a linear-time prophet-insertion is similar to the optimization techniques for the plain-insertion algorithm in Section 5.2. We first handle the special case (when $i = j$) in linear time and then apply **dynamic programming (DP)** to address the general case (when $i < j$). The *major difference* is that we need to consider the worker's *waiting time* in the prophet-insertion. This is because a worker can come to the (predicted) request's origin earlier before the request appears. For example, if the release time of a predicted request is 7:05 am, the worker can arrive at 7:00 am and wait 5 minutes until the request appears.

Let $\text{wait}[k]$ be the waiting time at the location l_k of the route S_w. Then we have:

$$\text{wait}[k] = \begin{cases} \max\{t_{r'} - \text{arr}[k], 0\}, & \text{if } l_k \text{ is the origin of a request } r' \\ 0, & \text{otherwise} \end{cases} \tag{21}$$

Hence the definition of $\text{arr}[\cdot]$ is changed into Equation (22).

$$\text{arr}[k+1] = \max\{\text{arr}[k], t_{r'}\} + \text{dis}(l_k, l_{k+1}) \tag{22}$$

The formulation is correct from two aspects: (1) $\text{dis}(l_k, l_{k+1})$ is the travel time between the locations l_k and l_{k+1} and (2) $\max\{\text{arr}[k], t_{r'}\}$ is the leaving time at the location l_k. Here, we use the max function since a worker may arrive at the origin l_k before the request r' appears at time $t_{r'}$.

We also use $\text{swait}[k]$ to denote the sum of waiting time from location l_k to location l_n, i.e., $\text{swait}[k] = \sum_{i=k}^{n} \text{wait}[i]$, which can be initialized as follows.

$$\text{swait}[k] = \begin{cases} 0, & \text{if } k = n+1, \\ \text{swait}[k+1] + \text{wait}[k], & \text{otherwise} \end{cases} \tag{23}$$

The waiting time also affects the concept $\text{slack}[\cdot]$ in Section 5.1, which represents the maximal tolerable time for detour between l_k and l_{k+1} to satisfy all the requests' deadlines after location l_k (excluded). That is, $\text{slack}[k]$ should be small enough to satisfy the inequalities below. Specifically, when waiting time all equals to 0 in the plain-insertion, $\text{slack}[k]$ is no larger than $\text{ddl}[k'] - \text{arr}[k']$ for each $k' > k$ in Equation (11). In the prophet-insertion, a worker may wait at l_{k+1} for some time until the request at l_{k+1} is released, so he can further detour a longer time between l_k and l_{k+1} and reduce the waiting time (up to 0) at l_{k+1}. This is why we add $\text{wait}[k+1]$ after $\text{ddl}[k+1] - \text{arr}[k+1]$ in the **right hand side (RHS)** of the first inequality. Similarly, a worker may further wait at l_{k+2} and hence we add $\text{wait}[k+1] + \text{wait}[k+2]$ after $\text{ddl}[k+2] - \text{arr}[k+2]$ in the RHS of the second inequality. Inductively, we can also infer the other inequalities.

$$\begin{aligned} \text{slack}[k] &\le \text{ddl}[k+1] - \text{arr}[k+1] + \text{wait}[k+1] \\ \text{slack}[k] &\le \text{ddl}[k+2] - \text{arr}[k+2] + \text{wait}[k+1] + \text{wait}[k+2] \\ &\cdots \\ \text{slack}[k] &\le \text{ddl}[n] - \text{arr}[n] + \text{wait}[k+1] + \cdots + \text{wait}[n] \end{aligned}$$

According to the inequalities above, $\text{slack}[k]$ is defined as

$$\text{slack}[k] = \min_{k+1 \le i \le n} \left\{ \text{ddl}[i] - \text{arr}[i] + \sum_{j=k+1}^{i} \text{wait}[j] \right\} \tag{24}$$

Based on the definition of slack$[k]$ in Equation (24), we also have

$$\text{slack}[k+1] = \min_{k+2 \le i \le n} \left\{ \text{ddl}[i] - \text{arr}[i] + \sum_{j=k+2}^{i} \text{wait}[j] \right\} \tag{25}$$

Thus, we can infer the DP equation of slack$[k]$ as follows.

$$\begin{aligned}
\text{slack}[k] &= \min_{k+1 \le i \le n} \left\{ \text{ddl}[i] - \text{arr}[i] + \sum_{j=k+1}^{i} \text{wait}[j] \right\} \\
&= \text{wait}[k+1] + \min_{k+1 \le i \le n} \left\{ \text{ddl}[i] - \text{arr}[i] + \sum_{j=k+2}^{i} \text{wait}[j] \right\} \\
&= \text{wait}[k+1] + \min \left\{ \text{ddl}[k+1] - \text{arr}[k+1], \min_{k+2 \le i \le n} \left\{ \text{ddl}[i] - \text{arr}[i] + \sum_{j=k+2}^{i} \text{wait}[j] \right\} \right\} \\
&= \text{wait}[k+1] + \min \{ \text{ddl}[k+1] - \text{arr}[k+1], \text{slack}[k+1] \}
\end{aligned} \tag{26}$$

Thus, the arrays of wait, arr, swait, slack are initialized in linear time by Equations (21), (22), (23), and (26), respectively. The other arrays, ddl and pick, are still initialized by Equations (9) and (12).

7.2.2 Handling Special Case $i = j$. Since the number of special cases ($i = j$ as in Figures 3(b) and 3(c)) is $O(n)$, we will process the special cases in linear time if we can *(1)* calculate the increased travel time $\Delta_{i,j}$ in $O(1)$ time, and *(2)* check the constraints in $O(1)$ time.

Calculating $\Delta_{i,j}$ in $O(1)$ Time. Since the worker may wait at the new request's origin, we first use wait_r to denote the waiting time at the origin in the following, i.e.,

$$\text{wait}_r = \max\{0, t_r - (\text{arr}[i] + \text{wait}[i] + \text{dis}(l_i, o_r))\} \tag{27}$$

Next, we show the calculation of the increased travel time $\Delta_{i,j}$ in $O(1)$ time from two cases.

- **Case $i = j = n$ in Figure 3(b).** In this case, the origin and destination of the new request r is appended at the end of the current route S_w. Thus, the increased travel time consists of three parts: the travel time from the end of S_w (i.e., location l_n) to the new request's origin o_r, the waiting time at o_r, and the travel time from the origin o_r to the destination d_r, i.e.,

$$\Delta_{i,j} = \text{dis}(l_n, o_r) + \text{wait}_r + \text{dis}(o_r, d_r) \quad (\text{when } i = j = n) \tag{28}$$

- **Case $i = j < n$ in Figure 3(c).** In this case, the origin and destination of the new request r are sequentially inserted between the locations l_i and l_{i+1}. Our idea is to first compute the detour between l_i and l_{i+1} and then infer the increased travel time ($\Delta_{i,j}$) of this insertion. Specifically, let detOD(i) denote the increased travel time between locations l_i and l_{i+1}:

$$\text{detOD}(i) = \text{dis}(l_i, o_r) + \text{wait}_r + \text{dis}(o_r, d_r) + \text{dis}(d_r, l_{i+1}) - \text{dis}(l_i, l_{i+1}) \tag{29}$$

  The sum of the first four terms is the travel time from l_i to l_{i+1} by the new route. Thus, the value of detour is obtained by subtracting the original travel time $\text{dis}(l_i, l_{i+1})$ from it. Accordingly, the increased travel time can be derived from Lemma 10.

  Lemma 10. *Given a function detOD(i) defined in Equation (29), we have*

$$\Delta_{i,j} = \max\{\textit{detOD}(i) - \textit{swait}[i+1], 0\} \quad (\textit{when } i = j < n) \tag{30}$$

Checking Constraints in $O(1)$ Time. We use Lemma 11 to check both capacity constraint and deadline constraint in $O(1)$ time.

Lemma 11. *The constraints are satisfied iff (1)* $pick[i] \le c_w - c_r$*, (2)* $detOD(i) \le slack[i]$*, and (3)* $arr[i] + wait[i] + dis(l_i, o_r) + wait_r + dis(o_r, d_r) \le e_r$*.*

7.2.3 Handling General Case $i < j$. In the general case in Figure 3(d), there are $O(n^2)$ possible pairs for (i, j), i.e., $i < j$ and $j = 1, \ldots, n$. Thus, we will only achieve quadratic time complexity if we can *(1)* calculate the increased travel time $\Delta_{i,j}$ in $O(1)$ time and *(2)* check the constraints in $O(1)$ time. To achieve linear time complexity, we also apply the dynamic programming techniques as mentioned in Section 5.2. Specifically, we also *(3)* enumerate the delivery location of the new request (i.e., $j = 1, \ldots, n$) in $O(n)$ time, and *(4)* find the best pickup location in $O(1)$ time, which has the minimum increased travel time for a fixed j. Details of these four steps are introduced as follows.

Step 1: Calculating $\Delta_{i,j}$ in $O(1)$ time. In the general case, the increased travel time involves two parts: the detour of inserting origin o_r after location l_i (denoted by $\mathrm{detO}(i)$) and the detour of inserting destination d_r after location l_j (denoted by $\mathrm{detD}(j)$). They are defined as follows, where wait_r represents the waiting time at the new request's origin as in Equation (27).

$$\mathrm{detO}(i) = \mathrm{dis}(l_i, o_r) + \mathrm{wait}_r + \mathrm{dis}(o_r, l_{i+1}) - \mathrm{dis}(l_i, l_{i+1}) \tag{31}$$

$$\mathrm{detD}(j) = \begin{cases} \mathrm{dis}(l_j, d_r) + \mathrm{dis}(d_r, l_{j+1}) - \mathrm{dis}(l_j, l_{j+1}) & \text{if } j < n \\ \mathrm{dis}(l_j, d_r) & \text{if } j = n \end{cases} \tag{32}$$

Accordingly, we can calculate the increased travel time $\Delta_{i,j}$ in $O(1)$ time by Lemma 12.

Lemma 12. *Given functions* $detO(i)$ *and* $detD(j)$ *defined in Equation (31) and (32), we have*

$$\Delta_{i,j} = \max\{detO(i) + detD(j) - swait[i+1], detD(j) - swait[j+1], 0\} \quad (when\ i < j)$$

Step 2: Checking Constraints in $O(1)$ Time. We use Lemma 13 to check both capacity constraint and deadline constraint in $O(1)$ time.

Lemma 13. *The constraints are satisfied iff (1)* $\forall k \in [i, j], pick[k] \le c_w - c_r$*, (2)* $detO(i) \le slack[i]$*, (3)* $detD(j) + \max\{detO(i) - (swait[i+1] - swait[j+1]), 0\} \le slack[j]$*, and (4)* $arr[j] + wait[j] + \max\{detO(i) - (swait[i+1] - swait[j+1]), 0\} + dis(l_j, d_r) \le e_r$*.*

Step 3: Enumerating Delivery Locations in $O(n)$ Time. Our linear-time prophet-insertion algorithm enumerates only the delivery location j in $O(n)$ time. Let Δ_j be the minimum increased travel time for a given j and Δ as the minimum increased travel time for all possible j, i.e.,

$$\Delta_j = \min_{i<j}\{\Delta_{i,j}\} \tag{33}$$

$$\Delta = \min_{0<j\le n} \min_{i<j}\{\Delta_{i,j}\} = \min_{0<j\le n}\{\Delta_j\} \tag{34}$$

Let $\mathrm{plc}[j]$ be the best place for inserting the new request's origin o_r (corresponding to Δ_j), i.e.,

$$\mathrm{plc}[j] = \arg\min_{i<j}\{\Delta_{i,j}\} \tag{35}$$

Next, we explain how to efficiently calculate Δ_j and $\mathrm{plc}[j]$ in $O(1)$ time.

Step 4: Finding the Best Pickup Location in $O(1)$ Time. Since the best pickup location $\mathrm{plc}[j]$ corresponds to the minimum increased travel time Δ_j for the given j, we first focus on efficiently calculating Δ_j in Lemma 14.

Lemma 14. *Given an array* $dio[j] = \min_{i<j}\{detO(i) - swait[i+1]\}$*, we can calculate the minimum increased travel time* Δ_j *for the given* j *in* $O(1)$ *time as*

$$\Delta_j = \max\{detD(j) + dio[j], detD(j) - swait[j+1], 0\} \tag{36}$$

if $dio[\cdot]$ *can be pre-processed in* $O(n)$ *time.*

Now we show how to pre-process dio[·] in Lemma 14 in $O(n)$ time. Since the conditions of the constraints in Lemma 13 involve the terms of i, let $\text{dio}[j] = \infty$ be the case when some constraints are violated if we insert o_r after the location l_i. Accordingly, we can initialize dio[j] by Equation (37).

$$\text{dio}[j] = \begin{cases} \infty, & \text{if } \text{pick}[j-1] > c_w - c_r \\ \text{dio}[j-1], & \text{if } \text{detO}(j-1) > \text{slack}[j-1] \\ \min\{\text{dio}[j-1], \text{detO}(j-1) - \text{swait}[j]\}, & \text{otherwise} \end{cases} \tag{37}$$

Correctness. We prove the correctness of Equation (37) as follows. The first case $\text{pick}[j-1] > c_w - c_r$ violates the capacity constraint (i.e., Lemma 13 (1)). Thus, $\text{dio}[j] = \infty$ because the origin o_r cannot be inserted after any location before l_{j-1} (including). The second case $\text{detO}(j-1) > \text{slack}[j-1]$ violates the deadline constraint (i.e., Lemma 13 (2)). It only indicates that the origin o_r cannot be inserted after l_{j-1}. Since someplace before l_{j-1} (excluded) may be still feasible to insert o_r, we set dio[j] as dio[$j-1$]. The last case can be derived by DP as follows.

$$\begin{aligned} \text{dio}[j] &= \min_{i<j}\{\text{detO}(i) - \text{swait}[i+1]\} \\ &= \min\left\{\min_{i<j-1}\{\text{detO}(i) - \text{swait}[i+1]\}, \text{detO}(j-1) - \text{swait}[j]\right\} \\ &= \min\{\text{dio}[j-1], \text{detO}(j-1) - \text{swait}[j]\} \end{aligned}$$

Based on Equation (37), the best pickup location plc[j] for the given j is:

$$\text{plc}[j] = \begin{cases} \text{not exist}, & \text{if } \text{pick}[j-1] > c_w - c_r \\ \text{plc}[j-1], & \text{if } \text{detO}(j-1) > \text{slack}[j-1] \\ \text{plc}[j-1], & \text{if } \text{dio}[j-1] < \text{detO}(j-1) - \text{swait}[j] \\ j-1, & \text{otherwise} \end{cases} \tag{38}$$

The correctness of Equation (38) is guaranteed by Lemma 15.

Lemma 15. *For the given j, if plc[j] violates constraints, then other $i \neq \text{plc}[j]$ also violates constraints.*

By the proof of Lemma 15 and the definitions of dio[·] and plc[·], we can simplify the tests of the constraints in Lemma 13 by Corollary 2.

Corollary 2. *For the given j, a feasible i for inserting o_r exists iff (1) $pick[j] \leq c_w - c_r$, (2) $detD(j) + \max\{dio[j] + swait[j+1], 0\} \leq slack[j]$, and (3) $arr[j] + wait[j] + \max\{dio[j] + swait[j+1], 0\} + dis(l_j, d_r) \leq e_r$.*

7.2.4 Putting It Together. Algorithm 7 illustrates our linear-time prophet-insertion algorithm. In line 1, we use S_w^* to denote the new route with the minimum increased travel time Δ after inserting the request r. In line 2, we perform the initialization of pick, ddl, arr, swait, slack, dio, plc by Equations (9), (12), (22), (23), (24), (37), and (38), respectively. In line 3, we determine the feasible ranges for inserting the new request's destination by our pruning strategy in Lemma 16, i.e., $j \in [begj, endj]$. In lines 4–7, we address the special case (when $i = j$). Specifically, for each possible j from $begj$ to $endj$, we first check the constraints by Lemma 11 (line 5), then calculate the increased travel time $\Delta_{i,j}$ by Equations (28) or (30) (line 6), and finally maintain Δ, i^*, j^* (line 7). In lines 8–11, we focus on the general case (when $i < j$). In line 8, we also enumerate each possible j from $begj$ to $endj$. For each j, we check the constraints in line 9 and calculate the minimum increased travel time Δ_j by Equation (36) in line 10. If Δ_j is smaller than Δ, we maintain the minimum increased travel time Δ and the best insertion places (i^*, j^*) for origin o_r and destination d_r in line 11. Finally,

Table 4. wait[·], swait[·], ddl[·], arr[·], pick[·], slack[·], dio[·] and plc[·] in Example 7

k	wait[k]	swait[k]	ddl[k]	arr[k]	pick[k]	slack[k]	dio[k]	plc[k]
0	0	5	0	0	0	18	∞	doesn't exist
1	5	5	18	0	1	13	12	0
2	0	0	28	15	0	∞	5	1

ALGORITHM 7: Linear-time prophet-insertion

input : a worker w with a route S_w and a request r
output: a new route S_w^* for the worker w

1 $S_w^* \leftarrow S_w, \Delta \leftarrow \infty$;
2 Initialize ddl, pick, arr, swait, slack, dio, plc;
3 $[begj, endj] \leftarrow$ the feasible range for j by Lemma 16;
4 **foreach** $j \leftarrow begj$ **to** $endj$ **do** /* handle the special case $i = j$ */
5 **if** *Lemma 11 is violated* **then** continue;
6 $\Delta_{i,j} \leftarrow$ calculate by Equations (28) or (30);
7 **if** $\Delta_{i,j} < \Delta$ **then** $\Delta, i^*, j^* \leftarrow \Delta_{i,j}, j, j$;
8 **foreach** $j \leftarrow begj$ **to** $endj$ **do** /* handle the general case $i < j$ */
9 **if** $j > 0$ *and Corollary 2 is satisfied* **then**
10 $\Delta_j \leftarrow$ calculate by Equation (36);
11 **if** $\Delta_j < \Delta$ **then** $\Delta, i^*, j^* \leftarrow \Delta_j$, plc[$j$], j;
12 **if** $\Delta < \infty$ **then** $S_w^* \leftarrow$ insert o_r and d_r after locations l_{i^*} and l_{j^*}, respectively;

we obtain the new route S_w^* with the minimum increased travel time Δ to insert the request r in line 12.

Optimization by Pruning. Since the route S_w may contain many predicted requests, we also propose pruning strategies to avoid impossible j for our linear-time prophet-insertion as follows.

Lemma 16. *Let r_{l_k} be the request of the location l_k in the route S_w and r be the new request:*

(1) If the deadline of r_{l_k} is earlier than the release time of r, we can safely prune any $j < k$.
(2) If the arrival time at l_k has exceeded the deadline of r, we can safely prune any $j \geq k$.

Example 7. Back to Example 6. A predicted request r_2 is inserted into a route $\langle(v_2, 0), (v_2, 5), (v_4, 0)\rangle$. We can use Algorithm 7 to calculate the minimum increased travel time. The arrays in line 2 of Algorithm 7 are initialized as in Table 4. Specifically, the waiting time wait[k] is calculated by Equation (21), e.g., wait[1] $= \max\{t_{r_1} - \text{arr}[1], 0\} = 5$. By Equation (23), we can calculate swait[k], e.g., swait[1] = wait[1] + swait[2] = 5. By Equations (9) and (12), we initialize ddl[·] and pick[·] as in Example 3. By Equation (22), we can initialize the array arr[k], e.g., arr[2] $= \max\{\text{arr}[1], t_{r_1}\} + \text{dis}(v_2, v_4) = \max\{0, 5\} + 10 = 15$. By Equation (26), we have slack[1] = wait[2] + $\min\{\text{ddl}[2] - \text{arr}[2], \text{slack}[2]\} = 0 + \min\{28 - 15, \infty\} = 13$, slack[0] = wait[1] + $\min\{\text{ddl}[1] - \text{arr}[1]$, slack[1]$\} = 5 + \min\{18 - 0, 13\} = 18$. Besides, dio[0] $= \infty$, plc[0] = not exists, since a feasible i cannot exist for $j = 0$ in the general case as shown in Figure 3(d). By using Equations (37) and (38), we have dio[1] = $\min\{\text{dio}[0], \text{detO}(0) - \text{swait}[1]\} = \min\{\infty, 17 - 5\} = 12$, plc[1] $= 1 - 1 = 0$, since $\text{detO}(0) = \text{dis}(v_2, o_{r_2}) + \text{wait}_{r_2} + dis(o_{r_2}, v_2) - \text{dis}(v_2, v_2) = 7 + 3 + 7 - 0 = 17 <$ slack[0] (i.e., the last case of Equation (37)) and dio[0] $>$ detO(0) − swait[1] (i.e., the last case of Equation (38)), where $\text{wait}_{r_2} = \max\{0, t_{r_2} - (\text{arr}[0] + \text{wait}[0] + \text{dis}(v_2, o_{r_2}))\} = \max\{0, 10 - (0 + 0 + 7)\} = 3$ by Equation (27). Similarly, we also have dio[2] = $\min\{\text{dio}[1], \text{detO}(1) - \text{swait}[2]\} = \min\{12, 5 - 0\} = 5$,

plc[2] = 2 − 1 = 1, since detO(1) = dis(v_2, o_{r_2}) + wait$_{r_2}$ + dis(o_{r_2}, v_4) − dis(v_2, v_4) = 7 + 0 + 8 − 10 = 5 < slack[1] (i.e., the last case of Equation (37)) and dio[1] > detO(1) − swait[2] (i.e., the last case of Equation (38)), where wait$_{r_2}$ = max{0, t_{r_2} − (arr[1] + wait[1] + dis(v_2, o_{r_2}))} = max{0, 10 − (0+5+7)} = 0 by Equation (27). In line 3, we have [*begj*, *endj*] = [0, 2] by Lemma 16.

In lines 4–7, we handle the **special case** $i = j$. When $j = 0$, we find that Lemma 11 (2) is violated in line 5, since detOD(0) = dis(v_2, o_{r_2}) + wait$_{r_2}$ + dis(o_{r_2}, d_{r_2}) + dis(d_{r_2}, v_2) − dis(v_2, v_2) = 7 + 3 + 10 + 10 − 0 = 30 > slack[0], where wait$_{r_2}$ = max{0, t_{r_2} − (arr[0] + wait[0] + dis(v_2, o_{r_2}))} = max{0, 10 − (0 + 0 + 7)} = 3 by Equation (27). When $j = 1$, Lemma 11 is satisfied. Specifically, Lemma 11 (1) checks the capacity constraint, which is obviously satisfied. To check Lemma 11 (2), we have detOD(1) = dis(v_2, o_{r_2})+wait$_{r_2}$+dis(o_{r_2}, d_{r_2})+dis(d_{r_2}, v_4)−dis(v_2, v_4) = 7+0+10+3−10 = 10 < slack[1], where wait$_{r_2}$ = max{0, t_{r_2}−(arr[1]+wait[1]+dis(v_2, o_{r_2}))} = max{0, 10−(0+5+7)} = 0 by Equation (27). As for Lemma 11 (3), we have arr[1]+wait[1]+dis(v_2, o_{r_2})+wait$_{r_2}$+dis(o_{r_2}, d_{r_2}) = 0+5+7+0+10 = 22, which is smaller than e_{r_2} = 31. In line 6, we calculate the increased travel time $\Delta_{i,j}$ = max{detOD(1) − swait[2], 0} = max{10 − 0, 0} = 10 by Equation (30). When the iterations in lines 4–7 stop, we have $\Delta = \Delta_{i,j} = 10, i^* = 1, j^* = 1$.

In lines 8–11, we handle the **general case** $i < j$. When $j = 1$, Corollary 2 (2) is violated in line 9, since detD(1) + max{dio[1] + swait[2], 0} = 3 + max{12 + 0, 0} = 15 > slack[1], where detD(1) = dis(v_2, d_{r_2}) + dis(d_{r_2}, v_4) − dis(v_2, v_4) = 10 + 3 − 10 = 3 by Equation (32). when j = 2, Corollary 2 is satisfied. For instance, Corollary 2 (2) is satisfied, since slack[2] = ∞. As for Corollary 2 (3), we have arr[2] + wait[2] + max{dio[2] + swait[3], 0} + dis(v_4, d_{r_2}) = 15 + 0 + max{5 + 0, 0} + 3 = 23, which is smaller than e_{r_2} = 31. In line 10, we calculate the increased travel time Δ_j. By Equation (36), we have Δ_2 = max{detD(2) + dio[2], detD(2) − swait[3], 0} = max{3 + 5, 3 − 0, 0} = 8 by Equation (36), where detD(2) = dis(v_4, d_{r_2}) = 3 by Equation (32). Since $\Delta_2 < \Delta$, we update $\Delta = \Delta_2 = 8, i^* = $ plc[2] $= 1, j^* = j = 2$ in line 11. Finally, we can obtain a new route $\langle (v_2, 0), (v_2, 5), (v_3(o_{r_2}), 0), (v_4, 0), (v_5(d_{r_2}), 0)\rangle$ in line 12, which is same as in Example 6.

Complexity Analysis. In Algorithm 7, lines 2–3, 4, 8, and 12 take $O(n)$ time and the other lines take $O(1)$ time. Thus, the time complexity of Algorithm 3 is $O(n)$ and space complexity is $O(n)$. If the shortest distance query takes $O(q)$ time, the algorithm takes $O(nq)$ time.

8 EXPERIMENTAL STUDY

This section presents the experimental evaluations of our proposed algorithms.

8.1 Experimental Setup

Datasets. We conduct experimental evaluations on two real citywide taxi datasets: *Chengdu* and *NYC*. The first is collected by Didi Chuxing in Chengdu, China, which is published through its GAIA initiative [11]. The second is a public dataset [37] collected from two types of taxis (yellow and green) in New York City, USA, and has been used in previous large-scale ride-sharing studies as benchmarks [1, 2, 43, 47]. As shown in Table 5, we randomly pick seven days of records to conduct the experiments. Each record includes the detailed information of each request (e.g., its release time, origin, and destination). Since only *NYC* contains the request size c_r, we generate c_r in *Chengdu* according to its distribution in *NYC*. Table 5 also reports the numbers of vertices and edges in each road network, which is downloaded from Geofabrik [21]. The numbers of vertices and edges in *NYC* are 43×-85× and 59×-78× larger than the road networks in [9, 25, 33, 54].

We simulate ride-sharing, a representative shared mobility application following the settings in [25, 33, 40, 54]. The origin and the destination of each request are mapped to the closest vertex in the road network. The initial location of a worker is randomly chosen from the vertices in the road network. Since a taxi usually travels at different speeds on different types of roads (e.g., 23

Table 5. Statistics of Datasets

Dataset	Collection date	#(Requests)	#(Vertices)	#(Edges)
NYC	April 8th-14th, 2016	392,157-517,850	5,270,965	11,151,356
Chengdu	November 2nd-8th, 2016	214,649-242,811	214,440	466,330

Table 6. Parameter Settings (the default settings are marked in bold)

Parameters	Settings
Number of workers $\|W\|$	500, 1000, 2000, **3000**, 4000, 5000
Worker's capacity c_w	3, **4**, 6, 10, 20
Delivery deadline e_r (second)	300, **600**, 900, 1200, 1500 ($+t_r + \text{dis}(o_r, d_r)$)
Weight α	0, **1**
Penalty coefficient β	10, 20, **30**, 40, 50
Penalty p_r	when $\alpha = \mathbf{1}$, $p_r = \beta\text{dis}(o_r, d_r)$; when $\alpha = 0$, $p_r = 1$

m/s in motorways or 6 m/s in residential streets), we assign a constant speed for each type of road (80% of the maximum legal speed limit in their cities). Table 6 summarizes the major parameters. Specifically, the delivery deadline is calculated as the parameter in the table added by the release time of the request and the travel time between its origin and destination. For example, the default deadline for a request r is $e_r = t_r + \text{dis}(o_r, d_r) + 600$. The parameter α in Equation (1) is 1 or 0. As explained in Section 3, when $\alpha = 0$ and each request's penalty $p_r = 1$, minimizing the unified cost is equivalent to the special case of maximizing the number of served requests. When $\alpha = 1$, the first term of the unified cost in Equation (1) is equivalent to the total travel time of workers, and the penalty p_r is calculated as the penalty coefficient β multiplied by the travel time between the origin o_r and the destination d_r, i.e., $p_r = \beta \times \text{dis}(o_r, d_r)$. We use $\beta = 10 \sim 50$ to represent the other special cases (i.e., maximizing the total revenue and minimizing the total travel time). The special case of minimizing the total travel time requires that β is a very large number. Here, we consider $\beta = 50$ as a large enough penalty coefficient, since it shows similar experimental patterns with a larger value (e.g., $\beta = 200$). Besides, we do not consider $\beta = 0$ in our experiments. When $\beta = 0$ and $\forall r, p_r = 0$, a simple solution, which rejects all the requests and plans an empty route for each worker, achieves the lowest unified cost (i.e., 0). We omit this case ($\beta = 0$) which has no penalty to reject a request, since it is impractical for real-world applications of shared mobility.

Compared Algorithms. In the following experiments, we compare our proposed algorithms **pruneGreedyDP (pGDP** for short) and Prophet with the state-of-the-art solutions.

- **T-share [33].** It filters workers via a bi-directional searching process and applies cubic-time insertion to find a worker with minimum increased distance for each online request.
- **Kinetic [25].** It uses a kinetic tree to maintain every possible route to serve all the remaining requests. Unlike T-share, the insertion operation in Kinetic takes quadratic time complexity.
- **PNAS [1].** It is a batch-based solution, so it periodically plans the set of requests that arrive within a specific time interval (e.g., 60 seconds in our experiments).
- **DAIF-DP* [54].** It is a demand-aware route planning algorithm, which applies quadratic-time insertion and prior knowledge of the requests. To demonstrate the superior performance of our algorithm, we let the prediction of DAIF-DP* be completely accurate in our experiments. Note that its effectiveness may become lower when the prediction is inaccurate.
- **pGDP.** It is the implementation of Algorithm 1 with linear-time plain-insertion.

- **Prophet.** It is the implementation of Algorithm 5 with linear-time prophet-insertion. Unlike DAIF-DP*, we do not assume prediction is accurate in Prophet. To justify the assumption of IID, we use past records as the prediction results (i.e., historical average [20]). For example, in the *NYC* dataset, we use the record collected on April 1st as the prediction of the testing data on April 8th, and so forth. In *Chengdu* dataset, we use the record collected on November 1st as the prediction of the testing data on November 2nd, and so forth.

Implementation. The experiments are conducted on a server with 40 Intel(R) Xeon(R) E5 2.30GHz processors with 128GB memory. The total running time for processing one-day records is limited to 24 hours, since a real-time solution should process all the requests before the time limitation. The shortest distance and shortest path queries are both on the fly, using the hub labeling based algorithm (SHP) proposed in [28]. An LRU cache with one million entries is maintained for the shortest distance and path queries by SHP in all the compared algorithms. Moreover, a grid index is also used in these algorithms, where the grid length is 1,000 meters by default. Under this setting, T-share needs at least 790GB spaces in *NYC* to maintain its data (e.g., spatially-ordered grid cell lists [33]) in the grid index, which is too large for a single server. Thus, we use a longer grid length (3,000 meters) for T-share in *NYC* dataset only. All the algorithms are implemented in GNU C++. Each experimental setting is repeated 7 times and the average results are reported.

Metrics. All the algorithms are evaluated in terms of *served rate* (i.e., $|R^+|/|R|$), *unified cost* (i.e., objective of the URPSM problem), *average response time* (i.e., average time to answer a ride-sharing request), *total running time* (i.e., total time to answer all the requests, update the index and maintain the workers' spatiotemporal data), and *memory usage*. All these five metrics are widely used in the related studies [25, 31–33, 58]. Note that the time to process the predictive requests is excluded in the total running time of Prophet, since they are pre-processed. Instead, we report the time cost of processing the predictive requests in Tables 7 and 8. For brevity, we use effectiveness to represent the performances of *served rate* and *unified cost* and use efficiency to represent the performances of *average response time*, *total running time*, and *memory usage*.

Except for these five mainstream metrics, we also consider two additional metrics: the (served) requests' average waiting time and the number of assigned requests to individual workers. Specifically, a request's waiting time is the duration from its release time to its completion time (i.e., when it is delivered at the destination). The number of assigned requests to a worker is the number of requests delivered by him on a daily average, which also indicates his potential salary.

8.2 Experimental Results

In the following, we present the results of the five major metrics in Section 8.2.1–8.2.6 and the results of the other two metrics in Section 8.2.7–8.2.8. Our experimental findings are summarized in Section 8.2.9. Due to page limitations, we omit some results which have similar patterns with the provided ones and please refer to our online appendix for the experiment on the prediction accuracy.

8.2.1 Impact of the Number of Workers $|W|$. Figure 4 presents the results of varying the number of workers. In terms of *effectiveness*, we can observe that our algorithm Prophet always outperforms the other algorithms. Specifically, the served rate of Prophet is up to 0.8×-14× and 13×-26× higher than the other algorithms in *Chengdu* and *NYC* datasets, respectively. In other words, when the number of workers is limited, Prophet can serve many more requests. The unified cost of Prophet can be up to 20× lower than the other algorithms in the *NYC* dataset. Among the other algorithms, the effectiveness of DAIF-DP* and pGDP are very close. For instance, the average difference of their served rate is less than 0.05% in either dataset and the unified cost of DAIF-DP* is only slightly lower than the unified cost of pGDP (e.g., no more than 0.0017× on average). Kinetic

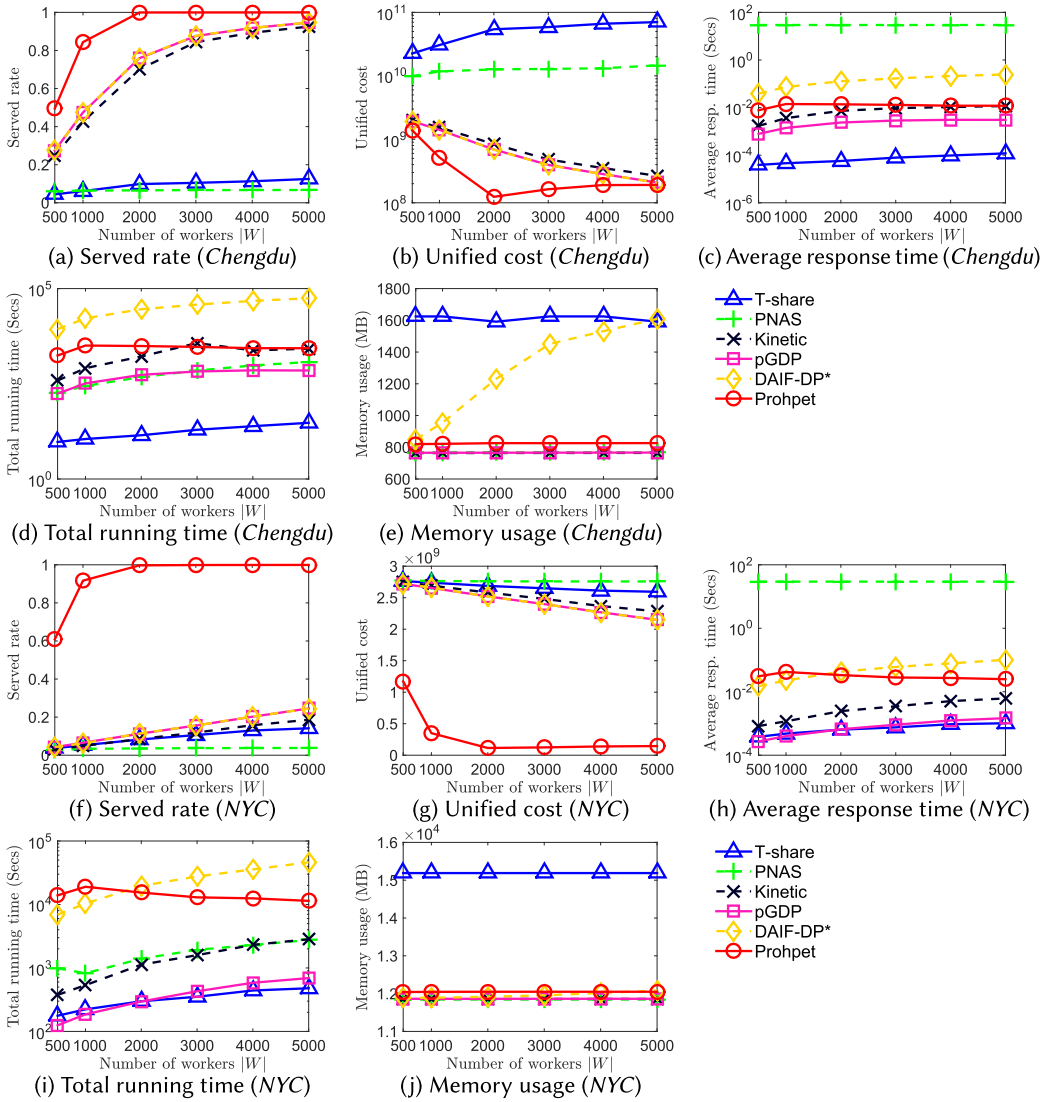


Fig. 4. Performance of varying the number of workers when $\alpha = 1$ in *Chengdu* and *NYC* datasets.

is less effective than DAIF-DP* and pGDP, while it is much more effective than T-share and PNAS. Besides, we observe the unified cost of Prophet first decreases and then increases. We also observe the served rate of Prophet has already approached to 100% when $|W|$ equals 2000. The variation pattern is reasonable, because the increased unified cost is due to the travel cost of the increased workers on their guidance routes. As for *average response time*, T-share is the most efficient and our algorithm pGDP is the runner-up. For example, the average response time of pGDP is up to 4× and 80× shorter than Kinetic and DAIF-DP*, respectively. Our algorithm Prophet is also a real-time solution, which is up to 20× faster than DAIF-DP*. PNAS is the only non-real-time solution. Since those requests released during every 60 seconds are processed together at the end of the 60 seconds, each request needs to wait for about 30 seconds before being processed by PNAS. In terms of *total running time*, T-share is the most efficient and our algorithm pGDP is often the

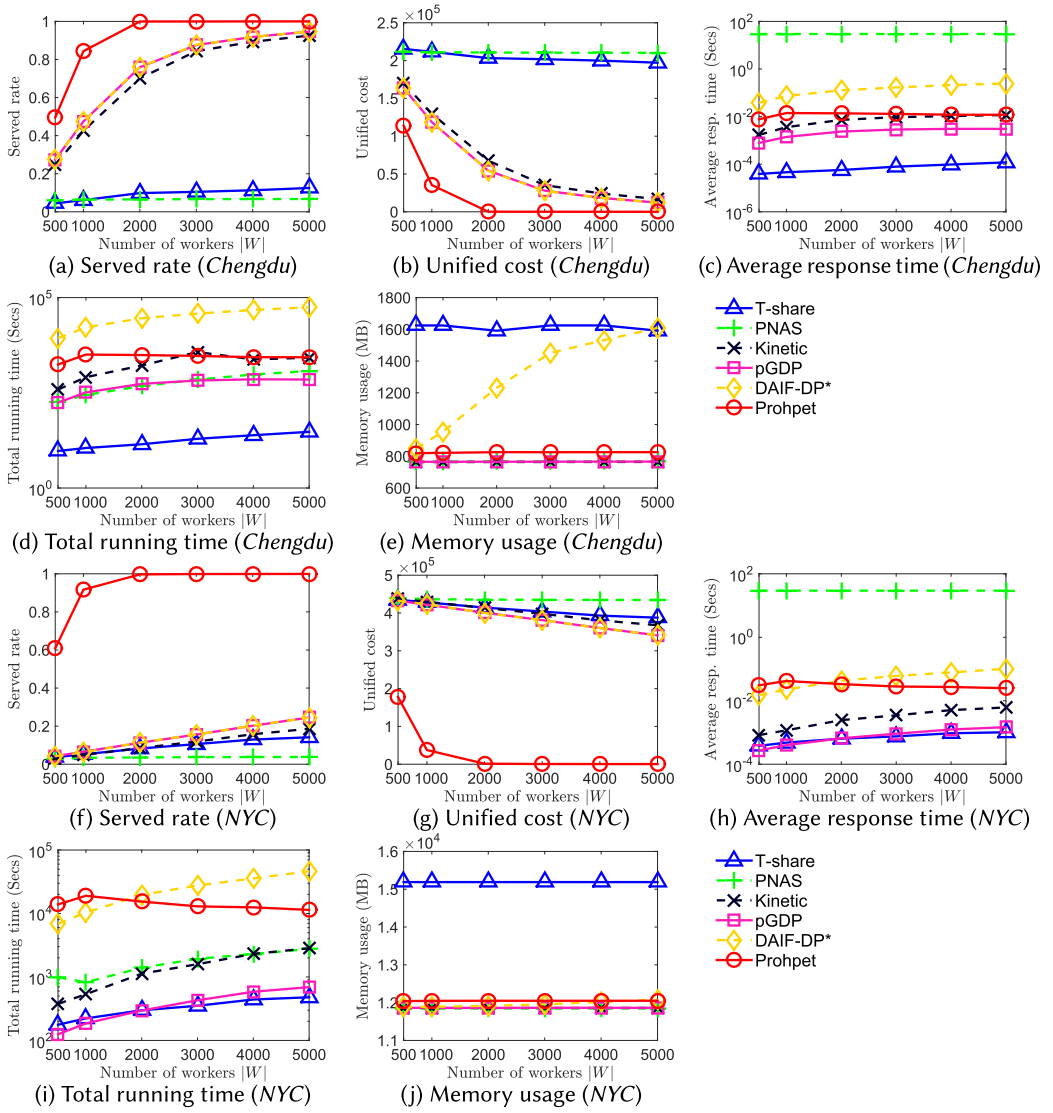


Fig. 5. Performance of varying the number of workers when $\alpha = 0$ in *Chengdu* and *NYC* datasets.

runner-up. For instance, in *NYC* dataset, the total running time of pGDP is up to 4×, 7×, and 66× shorter than Kinetic, PNAS and DAIF-DP*, respectively. DAIF-DP* is often the least efficient, since it takes a long time to maintain its data structures such as demand number map and total supply shift [54]. As for *memory usage*, all the algorithms are relatively efficient and T-share is the least efficient.

8.2.2 Impact of the Parameter α. Figure 5 presents the results of varying the number of workers when $\alpha = 0$. Compared with the previous results (when $\alpha = 1$) in Figure 4, we observe that the results of served rate, average response time, total running time and memory usage are barely changed. This is because all the compared algorithms still try to serve every request regardless of $\alpha = 0$ or 1. The main difference lies in the result of the unified cost. When $\alpha = 0$ and $\forall r \in R, p_r = 1$,

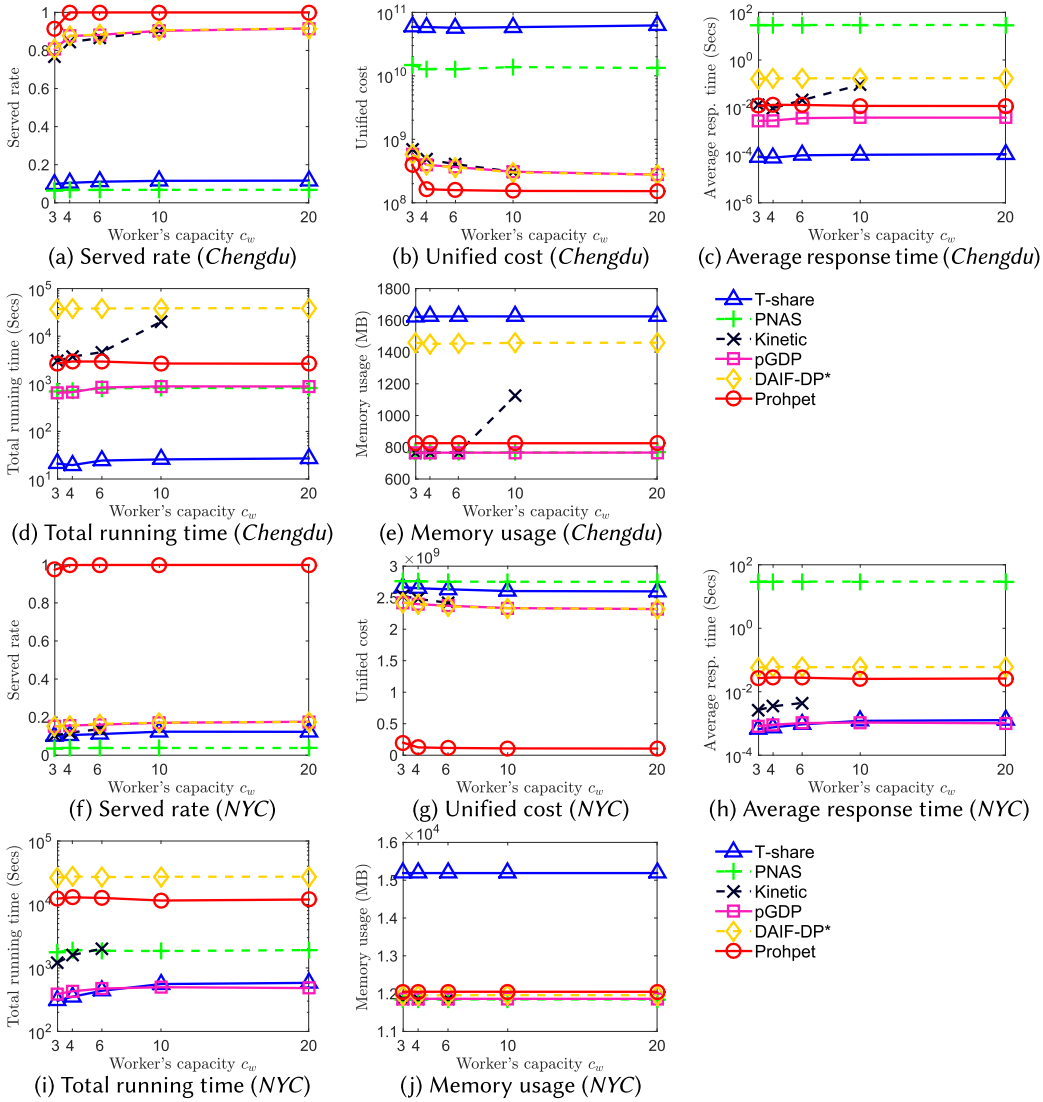


Fig. 6. Performance of varying the capacity of workers c_w in *Chengdu* and *NYC* datasets.

the value of unified cost equals to the number of the rejected requests (as explained in Section 3.2). By contrast, when $\alpha = 1$, the unified cost is the sum of the total travel cost and the total penalty of the rejected requests. Since the served rate of any algorithm does not change, we can also derive the results of the unified cost when $\alpha = 0$ from the results of served rate when $\alpha = 1$, i.e., unified cost = (1 − served rate) × number of requests. Since the number of requests in each dataset is fixed, a higher served rate when $\alpha = 1$ implies a lower unified cost under the case when $\alpha = 0$. Due to page limitations, we omit the results of varying the other parameters when $\alpha = 0$.

8.2.3 Impact of the Capacity of Workers c_w. Figure 6 shows the results of varying the capacity of workers. With a larger capacity, all the algorithms incur a higher *served rate* and a lower *unified cost* in both datasets. Our algorithm Prophet always has the highest served rate and the lowest unified cost. For instance, in the *NYC* dataset, the served rate of Prophet is at least 4.6× higher than the

others and the unified cost of Prophet is at least 11× lower than the others. The effectiveness of pGDP and DAIF-DP* are closed, which are worse than Prophet only. PNAS and T-share are notably less effective than others. For example, in the *Chengdu* dataset, PNAS has the lowest served rate and T-share has the highest unified cost. In terms of *average response time*, Kinetic becomes inefficient when the worker's capacity (i.e., c_w) increases. In our experiments, Kinetic could on average take longer than 5 seconds to process each request, when $c_w \geq 10$. As a result, Kinetic sometimes cannot be terminated in 24 hours, so we ignore the partial results of Kinetic. T-share is always the most efficient and pGDP is always the runner-up. For example, the average response time of pGDP is always shorter than 0.0038 seconds, which is up to 23× and 70× faster than Kinetic and DAIF-DP*, respectively. If the ignored cases (i.e., when $c_w \geq 10$ in *NYC*) are also considered, pGDP is up to 1320× faster than Kinetic. Our algorithm Prophet is relatively efficient, which often has a faster average response time than DAIF-DP* and PNAS. As for *total running time*, T-share is the most efficient and our algorithm pGDP is still the runner-up. Our algorithm Prophet is always faster than DAIF-DP*. In terms of *memory usage*, all the algorithms are efficient, except for Kinetic. In our experiments, when c_w increases to 20, Kinetic needs more than 24GB and 88GB spaces in the *Chengdu* and *NYC* datasets, respectively. In other words, Kinetic takes at least 30× more spaces than those of our algorithms (pGDP and Prophet). This is because each worker needs to store much more possible routes into the index (kinetic tree) of this baseline and consumes more spaces when the capacity increases to 20.

8.2.4 Impact of the Delivery Deadline e_r. Figure 7 illustrates the results of varying the delivery deadline e_r. With a larger deadline, the served rates of all the algorithms increase. The reason is that a longer deadline allows more requests to be served, and thus a higher served rate. In terms of *served rate* and *unified cost*, our algorithm (either Prophet or pGDP) achieves the best effectiveness. For instance, in *Chengdu* dataset, the served rate of Prophet is up to 0.6× higher than all the other algorithms. T-share and PNAS are usually less effective than others. We also observe the unified cost of Prophet becomes higher than pGDP when the varied parameter becomes larger than 900. This is because the delivery deadline has become long enough to serve almost all the requests under this setting and the guidance routes in Prophet lead to the increased cost. As for *average response time* and *total running time*, T-share is the most efficient and our algorithm pGDP is often the runner-up. In *NYC* dataset, pGDP is up to 5× and 71× faster than Kinetic and DAIF-DP* in terms of total running time, respectively. Our algorithm Prophet is more efficient than PNAS and DAIF-DP* in terms of average response time. Besides, DAIF-DP* cannot stop in 24 hours when the delivery deadline becomes long enough. As for *memory usage*, only DAIF-DP* consumes more spaces when the delivery deadline gets longer. This may be because the data structures (e.g., demand number map and total supply shift [54]) used in DAIF-DP* are sensitive to long delivery deadlines. The memory usages of the other algorithms are similar to the patterns in previous results.

8.2.5 Impact of the Penalty Coefficient β. Figure 8 presents the results of varying the penalty coefficient β. In terms of *served rate*, Prophet is always the highest, and pGDP and DAIF-DP* are the runner-ups. As for *unified cost*, the gaps between Prophet and others become larger when the penalty coefficient increases. For instance, in the *NYC* dataset, Prophet achieves the lowest unified cost, which is up to 30× and 31× lower than DAIF-DP* and Kinetic, respectively. In terms of *average response time* and *total running time*, T-share is the most efficient and either DAIF-DP* or PNAS is the least efficient. Our algorithm pGDP is always the runner-up in both datasets and Prophet is efficient enough to process each request in real-time. For example, in the *Chengdu* dataset, the average response time of pGDP is up to 3× shorter than that of Kinetic, and the average response time of Prophet is at least 12× shorter than that of DAIF-DP*. As for *memory usage*, the ranking of these algorithms in descending order is often T-share > DAIF-DP* > Prophet > Kinetic ≈ PNAS ≈ pGDP.

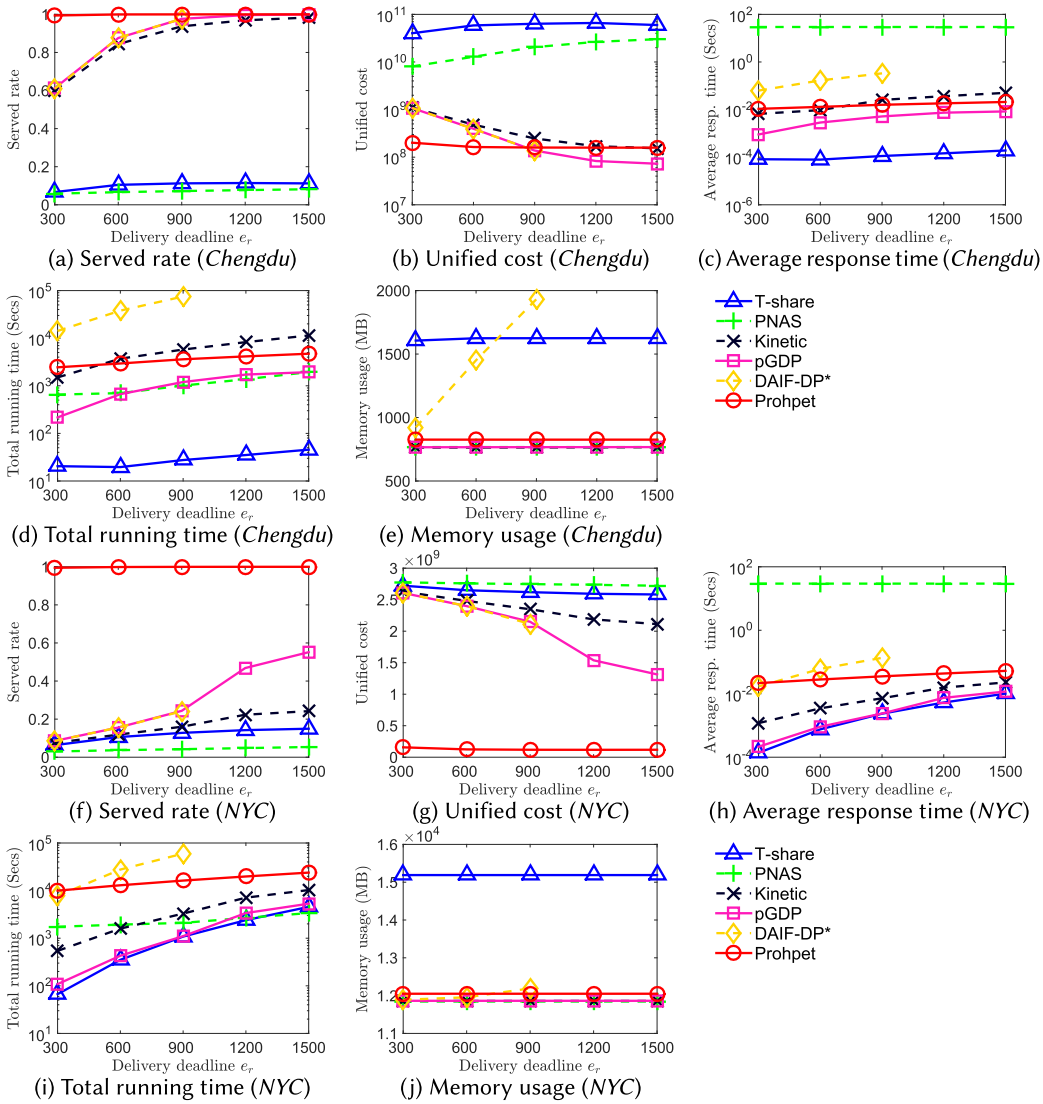


Fig. 7. Performance of varying the deadlines of requests e_r in *Chengdu* and *NYC* datasets.

Table 7. Total Running Time (Minute) for Processing the Predicted Requests in *Chengdu* Dataset

Number of requests	3 hours	6 hours	12 hours	18 hours	24 hours
Algorithm 4 + Algorithm 6	23.48	75.72	13626.46	>10 days	
Algorithm 4 + Algorithm 7	0.31	0.37	5.17	13.82	20.61

8.2.6 Impact of Different Prophet-Insertions. In Tables 7 and 8, we report the time cost of processing the predicted requests in Prophet under the default settings. Specifically, Algorithm 6 is cubic-time prophet-insertion. Algorithm 7 is linear-time prophet-insertion, which applies dynamic programming (DP). To demonstrate the scalability, we vary the number of requests (e.g., from

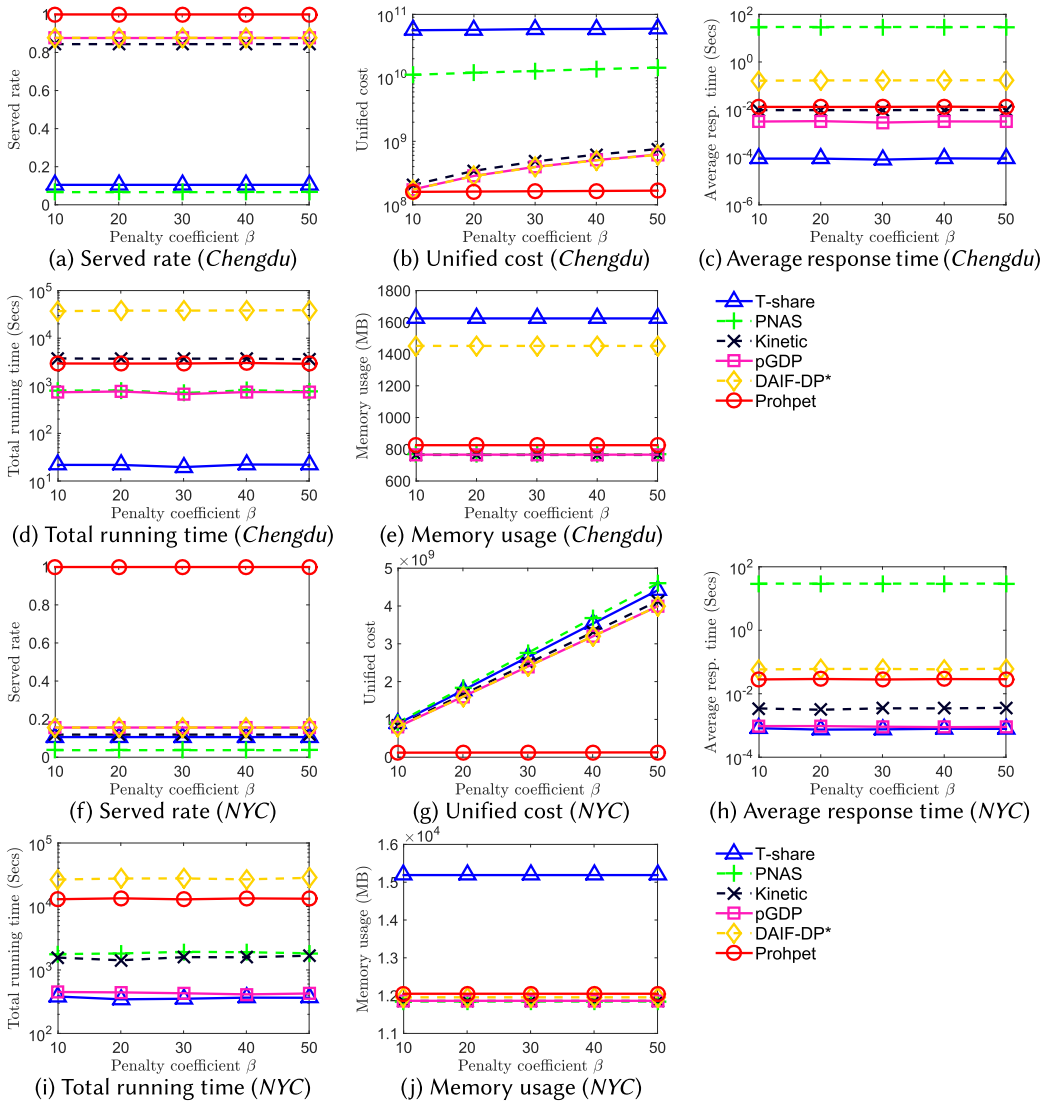
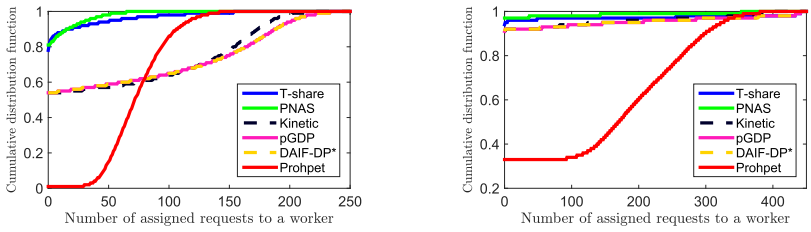


Fig. 8. Performance of varying the penalty coefficient β in *Chengdu* and *NYC* datasets.

Table 8. Total Running Time (Minute) for Processing the Predicted Requests in *NYC* Dataset

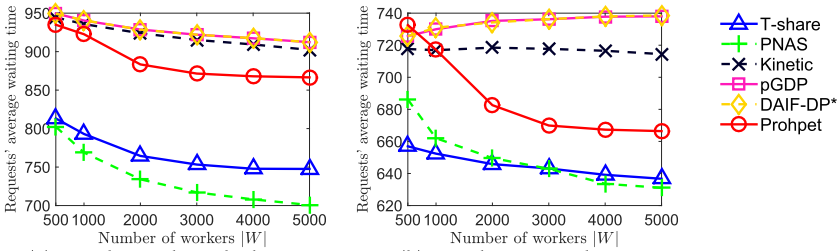
Number of requests	3 hours	6 hours	12 hours	18 hours	24 hours
Algorithm 4 + Algorithm 6	1114.42	7247.08	>10 days		
Algorithm 4 + Algorithm 7	7.03	8.60	33.44	59.20	108.67

3 hours of requests to 24 hours of requests under the default settings). Our algorithm can efficiently process the predicted requests and the proposed DP techniques are effective to improve the time efficiency. For instance, Algorithm 7 is faster than Algorithm 6 by 2-3 orders of magnitude. Besides, the additional memory usage of Algorithm 7 is always less than 25KB in our experiments.



(a) Result in *Chengdu* dataset (b) Result in *NYC* dataset

Fig. 9. Results of the number of assigned requests to individual workers (in the default setting).



(a) Result in *Chengdu* dataset (b) Result in *NYC* dataset

Fig. 10. The (served) requests' average waiting time in *Chengdu* and *NYC* datasets.

8.2.7 Results of the Number of Assigned Requests to Individual Workers. Figure 9 depicts the experimental results. Specifically, the x-coordinate denotes the number x of assigned requests to a worker, e.g., 0-250 in Figure 9(a). The y-coordinate represents the percentage of workers who are assigned with no more than x requests. For example, in Figure 9(a), nearly 80% of workers are assigned with no request by PNAS and T-share. In Figure 9(a), we can also observe that most of the workers are assigned with more than 50 requests by Prophet while more than half of the workers are assigned with fewer than 10 requests by the other algorithms. This is because (1) PNAS and T-share have a low served rate and (2) Kinetic, pGDP, and DAIF-DP* greedily assign each request to the worker who has the minimum increased travel cost to serve it, which makes workers in the areas with more requests get more assignments. This pattern indicates that at least 50% of the workers can potentially increase their salaries by using Prophet. In Figure 9(b), this pattern becomes even more notable in the *NYC* dataset, where over 60% of the workers can potentially increase their salaries by using Prophet. Although the number of workers assigned with over 200 requests by Prophet is smaller than that by pGDP, Kinetic, and DAIF-DP* in Figure 9(a), we still think Prophet is a better solution to the URPSM problem in terms of this metric due to these reasons: (1) Prophet has the highest served rate in all our tests; and (2) from the global point of view, many more workers can potentially increase their salaries by averagely serving more requests by Prophet than the other compared algorithms. Due to page limitations, we only report the results of our default settings here.

8.2.8 Results of the Requests' Average Waiting Time. Figure 10 presents the results of the (served) requests' average waiting time ("average waiting time" for short) when varying the number of workers. We can observe the average waiting time of T-share and PNAS is shorter than that of other algorithms. This is because T-share and PNAS serve much fewer requests than the others, which makes delivering a request take the shorter detour. Similarly, since Kinetic also serves fewer requests than DAIF-DP* and pGDP, the average waiting time of Kinetic is shorter than that of

DAIF-DP* and pGDP. In both datasets, Prophet often achieves a shorter average waiting time than Kinetic, DAIF-DP*, and pGDP, since the workers in Prophet are guided to where new requests may appear in advance. In Figure 10(b), we also observe that the average waiting time of DAIF-DP* and pGDP is increasing with the increase of workers. This is still reasonable, since they only consider the average waiting time in the constraint of a request's deadline. In other words, a request may take a longer detour to share with other requests, as long as it can be delivered before the deadline. Due to page limitations, we omit the results when varying the other parameters.

8.2.9 Summary of Results. We summarize our experimental findings as follows.

- Our algorithm is always the most effective. Specifically, Prophet has the highest served rate and lowest unified cost in most of the time (it has a lower unified cost than our algorithm pGDP only in Figure 7). For example, the served rate of Prophet is up to 13×-32× higher than the compared baselines. The unified cost of Prophet is up to 30×, 31×, 435×, and 188× lower than DAIF-DP*, Kinetic, T-share, and PNAS, respectively. The effectiveness of our algorithm pGDP is always better than Kinetic, T-share, and PNAS. For instance, the unified cost of pGDP is 412× lower than T-share and PNAS. The effectiveness of pGDP is also comparable with DAIF-DP*, as the difference of their served rates is only 0.03% on average.
- In terms of time efficiency, our algorithms, Prophet and pGDP, are both efficient enough to process the requests in real-time. In particular, pGDP is up to 23× and 80× faster than Kinetic and DAIF-DP* in terms of average response time, respectively. Although Prophet trades off the time efficiency for the significant improvements of effectiveness, it is still often much more efficient than DAIF-DP* and PNAS in terms of average response time.
- Considering the other two metrics in Section 8.2.7–8.2.8, our algorithm Prophet is effective to increase the average salary of each worker, since each worker is potentially assigned with more requests by Prophet than the other compared algorithms. In terms of the requests' average waiting time, Prophet is often better than Kinetic, DAIF-DP*, and pGDP.
- Among the baselines, the time efficiency of Kinetic and DAIF-DP* is not robust enough. For instance, Kinetic [25] cannot process the requests in real-time when the worker's capacity is large enough. DAIF-DP* [54] fails to halt in time when the request's deadline is long enough. Although PNAS [1] and T-share [33] are less effective than the others, they often have shorter total time running and shorter requests' average waiting time than Kinetic and DAIF-DP*.
- Our DP-based techniques can significantly improve the time efficiency for processing the requests by insertions in practice. For example, the linear-time prophet-insertion (Algorithm 7) is faster than cubic-time prophet-insertion (Algorithm 6) by 2-3 orders of magnitude.

9 CONCLUSION

In this paper, we propose the URPSM problem, a unified formulation of route planning for shared mobility. It provides a flexible multi-objective function where mainstream optimization goals in existing studies can be reduced to special cases of the URPSM problem. Since the plain-insertion is a basic yet inefficient operation in many existing works, we develop a novel dynamic programming based algorithm, which reduces the time complexity of plain-insertion from cubic or quadratic time to linear time. Then, we devise an effective and efficient framework leveraging the above DP-based plain-insertion algorithm to address the URPSM problem approximately. To further improve the effectiveness, we exploit how to take advantage of the prediction, when historical data is available. Specifically, we propose a new insertion operator called *prophet-insertion* to plan routes for both predictive requests and online requests. We devise a linear-time prophet-insertion and a prophet-insertion-based framework for the URPSM problem. We prove that there is no polynomial-time

algorithm with a constant *competitive ratio* to solve the URPSM problem and its variants proposed in previous studies. However, the *optimality ratio* of our prophet-insertion-based solution is a constant (0.47) to maximize the number of the served requests and total revenue of the platform. Extensive experiments on real datasets show that our solutions outperform the state-of-the-art algorithms in both effectiveness and efficiency by a large margin. Our paper serves as a comprehensive theoretical reference for route planning in shared mobility and opens up new opportunities for future research to design efficient solutions to large-scale shared mobility applications.

APPENDICES

A PROOFS IN PLAIN-INSERTION-BASED FRAMEWORK

This section lists the proofs of lemmas used in the plain-insertion algorithms (Section 5).

A.1 Proof of Lemma 4

Proof. As shown in Figure 3, condition (1) checks whether the deadline constraint of the new request r is violated, since $\mathrm{arr}[i] + \mathrm{dis}(l_i, o_r)$ represents the arrival time at the new request's origin (which is inserted after l_i). Condition (3) also checks whether the deadline constraint of the new request r is violated. For example, in Figures 3(b) and 3(c), $\mathrm{arr}[i] + \mathrm{dis}(l_i, o_r) + \mathrm{dis}(o_r, d_r)$ represents the arrival time at the new request's destination, which should be earlier than the deadline e_r.

Conditions (2) and (4) check whether the deadlines of the original requests are violated if o_r and d_r are inserted after l_i and l_j. For condition (2), the detour of inserting o_r after l_i should satisfy the deadlines of all the requests after l_i, i.e., the maximal tolerable time for *detour*. Thus, $\det(l_i, o_r, l_{i+1})$ should be smaller than $\mathrm{slack}[i]$. As for condition (4), the total *detour* to insert o_r and d_r should satisfy the deadlines of all the requests after l_j. Similarly, $\Delta_{i,j}$ should be smaller than $\mathrm{slack}[j]$.

A.2 Proof of Lemma 5

Proof. Condition (1) checks whether capacity constraint is violated if the new request is picked up after l_i. After the new request has been picked up, condition (2) checks whether capacity constraint is violated if the following original requests are served. Here we only need to check the original requests between l_i and l_j since the new request will be delivered after l_j.

A.3 Proof of Lemma 6

Proof. First, we assume $i = \mathrm{plc}[j]$ violates the capacity constraint (i.e., the first case of Equation (16) is satisfied). Based on Lemma 5, any other $i \le j - 1$ also violates the capacity constraint.

Next, we focus on the case when $i = \mathrm{plc}[j]$ only violates the deadline constraint in Lemma 4. Suppose to the contrary, there exists a z ($z < j$ and $z \neq \mathrm{plc}[j]$) which satisfies all the constraints. It indicates that for every $k \in [z, j)$, the capacity constraint $\mathrm{pick}[k] \le c_w - c_r$ always holds. Based on the definitions of $\mathrm{dio}[\cdot]$ and $\mathrm{plc}[\cdot]$, we have $\mathrm{dio}[j] = \det(l_i, o_r, l_{i+1}) \le \det(l_z, o_r, l_{z+1}) < \infty$ and the integer i must exist. We complete our proof based on the four conditions of Lemma 4.

When i violates Lemma 4 (1), it will also violate Lemma 4 (3), which will be proved later.

When i violates Lemma 4 (2), then we have $\det(l_i, o_r, l_{i+1}) > \mathrm{slack}[i]$. In other words, the second case of Equation (16) is satisfied and $\mathrm{plc}[i+1]$ should be equal to $\mathrm{plc}[i]$ instead of $(i+1) - 1 = i$. Thus, $\mathrm{plc}[j]$ cannot be equal to i, which is contradicted to the assumption.

When i violates Lemma 4 (3), then we have $\mathrm{arr}[j] + \det(l_i, o_r, l_{i+1}) + \mathrm{dis}(l_j, d_r) > e_r$, where $\det(l_i, o_r, l_{i+1})$ can be simplified by $\mathrm{dio}[j]$. Since $\det(l_i, o_r, l_{i+1}) \le \det(l_z, o_r, l_{z+1})$, then $\mathrm{arr}[j] + \det(l_z, o_r, l_{z+1}) + \mathrm{dis}(l_j, d_r)$ must be larger than e_r. Thus, z also violates Lemma 4 (3), which is contradicted to the assumption.

When i violates Lemma 4 (4), then we have $\Delta_{i,j} = \det(l_j, d_r, l_{j+1}) + \det(l_i, o_r, l_{i+1}) > \text{slack}[j]$, where $\det(l_i, o_r, l_{i+1})$ can be also simplified by $\text{dio}[j]$. Since $\det(l_i, o_r, l_{i+1}) \leq \det(l_z, o_r, l_{z+1})$, then $\Delta_{z,j} = \det(l_j, d_r, l_{j+1}) + \det(l_z, o_r, l_{z+1})$ must be larger than $\text{slack}[j]$. Thus, z also violates Lemma 4 (4), which is contradicted to the assumption.

A.4 Proof of Lemma 7

Proof. According to the definition, Δ is the actually increased time of worker w_i and $\Delta^{\downarrow}$ denotes the lower bound of the increased time of worker w_{i+1}. Since the workers are already sorted in ascending order by $\Delta^{\downarrow}$, it indicates that every worker after w_{i+1} has a longer increased time in terms of the lower bound. As the lower bound $\Delta^{\downarrow}$ of w_{i+1} is already longer than the actually increased time Δ of worker w_i, he/she must have a longer increased time than Δ, which can be pruned.

A.5 Proof of Lemma 8

Proof. We can prove the correctness of Equation (18) by substituting $\text{dis}(o_r, d_r) = \mathcal{L}$, $\text{dis}(l_i, l_{i+1}) = \text{len}[i+1]$ and $\forall u, v, \text{dis}^{\downarrow}(u, v) \leq \text{dis}(u, v)$ into the first two cases of Equation (8). We can prove the correctness of Equation (20) by substituting $\det^{\downarrow}(l_{j-1}, o_r, l_j) \leq \det(l_{j-1}, o_r, l_j)$ into Equation (15). We can prove the correctness of Equation (19) by substituting $\text{dio}^{\downarrow}[j] \leq \text{dio}[j]$ into Equation (14).

B PROOFS IN PROPHET-INSERTION-BASED FRAMEWORK

This section lists the proofs of lemmas and theorems used in the prophet-insertion-based framework (Section 6) and the prophet-insertion algorithms (Section 7).

B.1 Proof of Lemma 9

Proof. First, as mentioned in Section 6.2, the *plain-insertion* is a special case of the *prophet-insertion* used in Algorithm 4. In other words, if a request can be served by plain-insertion, it will also be able to be served by *prophet-insertion*. Moreover, since the *prophet-insertion* allows the workers to move to the request's origin earlier than its release time, it potentially serves more requests than *plain-insertion*.

Second, in Algorithm 4, we assume that there are sufficient hypothetical workers at the requests' origins (lines 1–6). In lines 7–11, we ensure that the real workers are assigned with the top $\{|W|\}$ most profitable routes by a max-heap, where the profit of a route is either the number of served request or the revenue of this route. Thus, each real worker is assigned to the currently most profitable route (i.e., the guidance route of the hypothetical worker $\widetilde{w^*}$). Suppose to the contrary, a real worker w is assigned to $\widetilde{S_w}$, which is more profitable than the route of $\widetilde{w^*}$ by Algorithm 4. Then, there must be a hypothetical worker $\widetilde{w} \neq \widetilde{w^*}$ corresponding to $\widetilde{S_w}$. Based on the definition of a max-heap, $\widetilde{w}$ must be popped earlier than $\widetilde{w^*}$ in line 9. However, since $\widetilde{w}$ is not matched to the real worker w in Algorithm 4, then some other worker must serve the requests in $\widetilde{w}$ with shorter increased travel time in line 9. Since insertion-based online algorithm always assigns the request to the worker with minimum increased travel time, this is contradicted to the prerequisite.

Finally, when the prediction is completely accurate, we can use Algorithm 4 to obtain the upper bound of any insertion-based online algorithm (only in the proof below).

B.2 Proof of Theorem 3

Proof. For brevity, we use OPT to denote the optimal insertion-based online algorithm and UB to denote the upper bound of OPT obtained by Algorithm 4. We also use ALG to denote our

framework (i.e., Algorithm 5). We will prove the (expected) optimality ratio E[ALG]/E[OPT] is a constant and the probability of achieving this ratio (0.47) is high in practice.

We use n and m to denote the number of online requests and predicted requests, respectively. W.l.o.g., we let $n = k \cdot m$. Under the assumption of known IID, these m predicted requests (a.k.a., request types) are sampled based on the distribution of the online requests in Algorithm 5. As prediction can be wrong, each online request has the probability of $\frac{1}{m}$ to match with a certain request type. Thus, each predicted request has the following probability to appear in the online requests.

$$1 - \left(1 - \frac{1}{m}\right)^n = 1 - \left(1 - \frac{1}{m}\right)^{k \cdot m} = 1 - \left(\frac{1}{e}\right)^k = 1 - \frac{1}{e^k} \tag{39}$$

Based on the proof of Lemma 9, we have E[OPT] $\leq$ E[UB]. We next prove the optimality ratio of Algorithm 5 in terms of each objective as follows.

Maximizing the number of served requests. Let $x_i \in \{0, 1\}$ be a random variable that represents the online request r_i is served in UB and $X = \sum_i x_i$ be a random variable that represents the number of served requests in UB. As E[OPT] $\leq$ E[UB], we can infer that E[UB] = E[X] = E[$\sum_i x_i$] $\geq$ E[OPT]. Let $\widetilde{x_i} \in \{0, 1\}$ be a random variable that represents the online request r_i is served in ALG and $\widetilde{X} = \sum_i \widetilde{x_i}$ be a random variable that represents the number of served requests in ALG. Based on the probability in Equation (39), we have

$$\text{E[ALG]} = \text{E}[\widetilde{X}] = \text{E}\left[\sum_i \widetilde{x_i}\right] = \text{E}\left[\left(1 - \frac{1}{e^k}\right)\sum_i x_i\right] = \left(1 - \frac{1}{e^k}\right)\text{E}\left[\sum_i x_i\right] \geq \left(1 - \frac{1}{e^k}\right)\text{E[OPT]} \tag{40}$$

Thus, the (expected) optimality ratio is $1 - \frac{1}{e^k}$. Here, we assume $k \in [0.65, 1.35]$ based on the experimental results of the prediction methods. Specifically, since the MAPEs of all the compared methods in [20] which predict the number of future requests are lower than 35%, the value of k is $[1 - 0.35, 1 + 0.35] = [0.65, 1.35]$. The optimality ratio is at least 0.47 based on this choice of k. We can derive the probability of getting this ratio by Azuma-Hoeffding inequality [35] as follows.

$$\text{for any } \epsilon > 0,\ \Pr\left[|\text{ALG} - \text{E[ALG]}| \geq \epsilon n\right] \leq 2e^{-\frac{(\epsilon n)^2}{2n}} = 2e^{-\epsilon^2 n/2} \tag{41}$$

In other words, with a probability at least $1 - e^{-\Omega(n)}$, the optimality ratio is 0.47 in practice.

Maximizing the total revenue. Let $y_i = x_i \cdot rev_{r_i}$ be a random variable that represents the revenue of the online request r_i if it is served in UB, where rev_{r_i} denotes the fare from serving the request minus the income of the worker. Let $Y = \sum_i y_i$ be a random variable that represents the total revenue in UB. As E[OPT] $\leq$ E[UB], we can infer that E[UB] = E[Y] = E[$\sum_i y_i$] $\geq$ E[OPT]. Let $\widetilde{y_i} = \widetilde{x_i} \cdot rev_{r_i}$ be a random variable that represents the revenue of the online request r_i if it is served in ALG and $\widetilde{Y} = \sum_i \widetilde{y_i}$ be a random variable that represents the total revenue in ALG. Similar to the deduction in Equation (40), we can infer that

$$\text{E[ALG]} = \text{E}[\widetilde{Y}] = \text{E}\left[\sum_i (\widetilde{x_i} \cdot rev_{r_i})\right] = \left(1 - \frac{1}{e^k}\right)\text{E}\left[\sum_i (x_i \cdot rev_{r_i})\right] \geq \left(1 - \frac{1}{e^k}\right)\text{E[OPT]} \tag{42}$$

Thus, the optimality ratio is $1 - \frac{1}{e^k}$ (i.e., at least 0.47 when $k \in [0.65, 1.35]$). To infer the probability of achieving this optimality ratio, we assume that $|rev_{r_i} - rev_{r_j}| \leq C$ for any two requests r_i and r_j, where C can be any large constant. Here, we only require C is still bounded when n approaches infinity. The assumption is reasonable since the fare of each request is much smaller than $O(n)$ in shared mobility. By Azuma-Hoeffding inequality [35], we infer the probability as:

$$\text{for any } \epsilon > 0,\ \Pr[|\text{ALG} - \text{E[ALG]}| \geq \epsilon n] \leq 2e^{-\frac{(\epsilon n)^2}{2nC^2}} = 2e^{-\epsilon^2 n/2C^2} \tag{43}$$

Therefore, with a high probability (at least $1 - e^{-\Omega(n)}$), the optimality ratio is 0.47.

B.3 Proof of Lemma 10

Proof. W.l.o.g., we assume the unit of travel time is second. By Equation (29), we have the following facts: (1) the worker will be $\text{detOD}(i)$ seconds late at the location l_{i+1}; and (2) he now needs less time (i.e., $\max\{\text{wait}[i+1] - \text{detOD}(i), 0\}$) to wait at the location l_{i+1}. Similarly, he may also take a longer time to arrive at any location after l_{i+1} and wait less time there. Based on the facts, we prove Lemma 10 from the following two aspects.

If $\text{detOD}(i) > \text{swait}[i+1]$, we will prove $\Delta_{i,j} = \text{detOD}(i) - \text{swait}[i+1]$, where $\text{swait}[i+1]$ is defined as $\text{wait}[i+1] + \cdots + \text{wait}[n]$. Based on the facts above, the worker does not need to wait at location l_{i+1} any more, i.e., $\max\{\text{wait}[i+1] - \text{detOD}(i), 0\} = 0$. Similarly, he will be $(\text{detOD}(i) - \text{wait}[i+1])$ seconds late at the location l_{i+2} and does not need to wait there any more since $\max\{\text{wait}[i+2] - (\text{detOD}(i) - \text{wait}[i+1]), 0\} = 0$. Inductively, he will be $(\text{detOD}(i) - \text{wait}[i+1] - \cdots - \text{wait}[n-1])$ seconds late at the location l_n. Since the last location l_n of any route is either a request's destination or a worker's initial location, he does not need to wait at l_n (i.e., $\text{wait}[n] = 0$ by Equation (21)). Thus, the increased travel time is exactly $\text{detOD}(i) - \text{wait}[i+1] - \cdots - \text{wait}[n-1] - \text{wait}[n] = \text{detOD}(i) - \text{swait}[i+1]$.

If $\text{detOD}(i) \le \text{swait}[i+1]$, we will prove $\Delta_{i,j} = 0$. Since $\text{detOD}(i) \le \text{swait}[i+1]$, there exists at least one integer $k \in [i+1, n]$ such that $\text{detOD}(i) \le \text{wait}[i+1] + \cdots + \text{wait}[k]$. We also assume k is the smallest among such integers. Based on the analysis above, we know the worker will be $(\text{detOD}(i) - \text{wait}[i+1] - \cdots - \text{wait}[k-1])$ seconds late at the location l_k, and only has to wait for $(\text{wait}[k] - (\text{detOD}(i) - \text{wait}[i+1] - \cdots - \text{wait}[k-1])) \ge 0$ seconds at the location l_k. After that, he will arrive at the location l_{k+1} exactly the same time as the original route before insertion. In other words, the total travel time will not be increased, i.e., $\Delta_{i,j} = 0$.

B.4 Proof of Lemma 11

Proof. Condition (1) checks whether the capacity constraint is violated if the new request r is inserted, which can be directly derived from Lemma 5.

Condition (2) checks whether the deadlines of the original requests are violated if o_r and d_r are sequentially inserted between l_i and l_{i+1}. Since the new request is inserted after l_i, the deadlines ($\text{ddl}[\cdot]$) before the location l_i (including) will not be violated. Thus, we only need to verify whether any deadline after the location l_i (excluded) is violated. Based on the definition of $\text{slack}[i]$, we only need to make sure that $\text{detOD}(i) \le \text{slack}[i]$.

Condition (3) checks whether the new request's deadline is violated. Since the LHS of this condition represents the arrival time at the new request's destination (as in Figures 3(b) and 3(c)), we only need to check whether it is no larger than the new request's deadline (i.e., e_r).

B.5 Proof of Lemma 12

Proof. By the proof of Lemma 10, we know the worker will be $\max\{\text{detO}(i) - \text{wait}[i+1] - \cdots - \text{wait}[k], 0\}$ seconds late at any location l_k after l_{i+1} (i.e., $k \ge i+1$). In other words, he will be $\max\{\text{detO}(i) - \text{wait}[i+1] - \cdots - \text{wait}[j], 0\}$ seconds late at the location l_j. As the new request's destination is inserted between l_j and l_{j+1}, he may be further $\text{detD}(j)$ seconds late at the location l_{j+1}, where $\text{detD}(j)$ is defined as the detour between l_j and l_{j+1} in Equation (32). Therefore, the late arrival time at the location l_n is

$$\begin{aligned}
&\max\left\{\max\{\text{detO}(i) - \text{wait}[i+1] - \cdots - \text{wait}[j], 0\} + \text{detD}(j) - \text{wait}[j+1] \cdots - \text{wait}[n], 0\right\} \\
&= \max\{\text{detO}(i) - \text{wait}[i+1] - \cdots - \text{wait}[n] + \text{detD}(j), \text{detD}(j) - \text{wait}[j+1] \cdots - \text{wait}[n], 0\} \\
&= \max\{\text{detO}(i) + \text{detD}(j) - \text{swait}[i+1], \text{detD}(j) - \text{swait}[j+1], 0\},
\end{aligned}$$

where $\text{swait}[k]$ is defined as $\text{wait}[k] + \cdots + \text{wait}[n]$ in Equation (23).

B.6 Proof of Lemma 13

Proof. Condition (1) checks the capacity constraint, which can be directly derived from Lemma 5.

Condition (2) checks whether the deadlines of the original requests are violated if the new request's origin is inserted between locations l_i and l_{i+1}. Since the deadlines before the location l_i are unaffected, we only need to verify whether any deadline after the location l_i (excluded) is violated. Based on the definition of slack$[i]$, we only need to ensure that detO$(i) \le$ slack$[i]$.

Condition (3) checks whether the deadlines of the original requests are violated if the new request's destination is inserted between locations l_j and l_{j+1}. Similarly, we only need to verify whether any deadline after the location l_j (excluded) is violated. Specifically, based on the proof of Lemma 12, the worker will be $\max\{\text{detO}(i) - \text{wait}[i+1] - \cdots - \text{wait}[j], 0\}$ seconds late at the location l_j. Based on the definition of swait$[\cdot]$, we have $\text{wait}[i+1] + \cdots + \text{wait}[j] = \text{swait}[i+1] - \text{swait}[j+1]$ and hence the late arrival time at the location l_j also equals to $\max\{\text{detO}(i) - (\text{swait}[i+1] - \text{swait}[j+1]), 0\}$. Thus, the late arrival time at the location l_{j+1} is $\text{detD}(j) + \max\{\text{detO}(i) - (\text{swait}[i+1] - \text{swait}[j+1]), 0\}$. Based on the definition of slack$[j]$, the late arrival time at the location l_{j+1} should be no larger than slack$[j]$.

Condition (4) checks whether the new request's deadline is violated. Specifically, the sum of the first three terms in the LHS of this condition represents the leaving time from the location l_j. Thus, the LHS of this condition represents the arrival time at the new request's destination. Here, we only need to check whether it is no larger than the new request's deadline (i.e., e_r).

B.7 Proof of Lemma 14

Proof. By substituting $\Delta_{i,j}$ (in Lemma 12) into Equation (33), we can derive that

$$\Delta_j = \min_{i<j} \Delta_{i,j} = \min_{i<j} \max\{\text{detO}(i) + \text{detD}(j) - \text{swait}[i+1], \text{detD}(j) - \text{swait}[j+1], 0\} \tag{44}$$

In Equation (44), since j is fixed, the terms related to only j are constants, e.g., detD(j), swait$[j+1]$. Thus, we can rewrite Equation (44) by Equation (45).

$$\Delta_j = \min_{i<j} \max\{F(i), C\} \tag{45}$$

where the function $F(i)$ and constant C are as follows (for proof brevity only):

$$F(i) = \text{detO}(i) - \text{swait}[i+1] + \text{detD}(j) \tag{46}$$

$$C = \max\{\text{detD}(j) - \text{swait}[j+1], 0\} \tag{47}$$

We next prove the following fact in Equation (48) from two aspects.

$$\min_{i<j} \max\{F(i), C\} = \max\{\min_{i<j}\{F(i)\}, C\} \tag{48}$$

- $\exists i, F(i) \le C$. We have both LHS and RHS of Equation (48) equal to C, since $\min_{i<j}\{F(i)\} \le C$.
- $\nexists i, F(i) \le C$. In other words, $\forall i, F(i) > C$. Thus, the LHS of Equation (48) equals $\min_{i<j}\{F(i)\}$. The RHS of Equation (48) also equals $\min_{i<j}\{F(i)\}$, since $\min_{i<j}\{F(i)\}$ is still larger than C.

Based on Equation (48), we can rewrite Equation (45) as follows:

$$\Delta_j = \min_{i<j} \max\{F(i), C\} = \max\{\min_{i<j}\{F(i)\}, C\} \tag{49}$$

As a result, we then focus on efficiently calculating $F(i)$. Since $\text{detD}(j)$ is a constant and $\text{dio}[j] = \min_{i<j}\{\text{detO}(i) - \text{swait}[i+1]\}$, we can calculate $\min_{i<j}\{F(i)\}$ as follows.

$$\begin{aligned}\min_{i<j}\{F(i)\} &= \min_{i<j}\{\text{detO}(i) - \text{swait}[i+1] + \text{detD}(j)\} \\ &= \text{detD}(j) + \min_{i<j}\{\text{detO}(i) - \text{swait}[i+1]\} \\ &= \text{detD}(j) + \text{dio}[j] \end{aligned} \tag{50}$$

Finally, we prove this lemma by substituting Equation (47) and (50) into Equation (49).

$$\Delta_j = \max\{\text{detD}(j) + \text{dio}[j], C\} = \max\{\text{detD}(j) + \text{dio}[j], \text{detD}(j) - \text{swait}[j+1], 0\}$$

B.8 Proof of Lemma 15

Proof. First, we assume $i = \text{plc}[j]$ violates the capacity constraint (i.e., the first case of Equation (38) is satisfied). By Lemma 13 (1), any other $i \le j-1$ also violates the capacity constraint.

Next, we focus on the case when $i = \text{plc}[j]$ only violates the deadline constraint in Lemma 13. Suppose to the contrary, there exists a z ($z < j$ and $z \neq \text{plc}[j]$) which satisfies all the constraints. It indicates that for every $k \in [z, j)$, the capacity constraint $\text{pick}[k] \le c_w - c_r$ always holds. Based on the definitions of $\text{dio}[\cdot]$ and $\text{plc}[\cdot]$, we have $\text{dio}[j] = \text{detO}(i) - \text{swait}[i+1] \le \text{detO}(z) - \text{swait}[z+1] < \infty$ and the integer i must exist. We complete our proof from the following three aspects.

When i violates Lemma 13 (2), we have $\text{detO}(i) > \text{slack}[i]$. In other words, the second case of Equation (38) is satisfied and $\text{plc}[i+1]$ should be equal to $\text{plc}[i]$ instead of $(i+1)-1 = i$. Thus, $\text{plc}[j]$ cannot be equal to i, which is contradicted to the assumption.

When i violates Lemma 13 (3), we have $\text{detD}(j) + \max\{\text{detO}(i) - \text{swait}[i+1] + \text{swait}[j+1], 0\} > \text{slack}[j]$, where $\text{detO}(i) - \text{swait}[i+1]$ can be simplified by $\text{dio}[j]$. Since $\text{detO}(i) - \text{swait}[i+1] \le \text{detO}(z) - \text{swait}[z+1]$, we can infer that $\text{detD}(j) + \max\{\text{detO}(z) - \text{swait}[z+1] + \text{swait}[j+1], 0\}$ is larger than $\text{slack}[j]$. Thus, z violates Lemma 13 (3), which is contradicted to the assumption.

When i violates Lemma 13 (4), we have $\text{arr}[j] + \text{wait}[j] + \max\{\text{detO}(i) - \text{swait}[i+1] + \text{swait}[j+1], 0\} + \text{dis}(l_j, d_r) > e_r$ and $\text{detO}(i) - \text{swait}[i+1]$ can be simplified by $\text{dio}[j]$. As $\text{detO}(i) - \text{swait}[i+1] \le \text{detO}(z) - \text{swait}[z+1]$, we derive $\text{arr}[j] + \text{wait}[j] + \max\{\text{detO}(z) - \text{swait}[z+1] + \text{swait}[j+1], 0\} + \text{dis}(l_j, d_r) > e_r$. Thus, z violates Lemma 13 (4), which is contradicted to the assumption.

B.9 Proof of Lemma 16

Proof. We prove (1) by assuming the contrary. If the new request's destination is feasible to be inserted after a location l_j, where $j < k$. Then the arrival time at the location l_k must be later than the release time of the request. By the prerequisite, we know the deadline of r_{l_k} must be violated, which is contradicted to the assumption.

We prove (2) by the prerequisite. We know $\text{arr}[j] > e_r$. Thus, the new request's deadline must be violated when its destination is inserted after location l_j. By the definition of $\text{arr}[\cdot]$ in Equation (22), we know $\text{arr}[j] \le \text{arr}[j+1]$. Thus, we can safely prune any $j \ge k$.

ACKNOWLEDGMENTS

We are grateful to the reviewers for their insightful and helpful comments.

REFERENCES

[1] Javier Alonso-Mora, Samitha Samaranayake, Alex Wallar, Emilio Frazzoli, and Daniela Rus. 2017. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proc. Natl. Acad. Sci.* 114, 3 (2017), 462–467.

[2] Mohammad Asghari, Dingxiong Deng, Cyrus Shahabi, Ugur Demiryurek, and Yaguang Li. 2016. Price-aware real-time ride-sharing at scale: An auction-based approach. In *SIGSPATIAL*. 3:1–3:10.

[3] Mohammad Asghari and Cyrus Shahabi. 2017. An on-line truthful and individually rational pricing mechanism for ride-sharing. In *SIGSPATIAL*. 7:1–7:10.
[4] Xiaohui Bei and Shengyu Zhang. 2018. Algorithms for trip-vehicle assignment in ride-sharing. In *AAAI*. 3–9.
[5] Allan Borodin and Ran El-Yaniv. 2005. *Online Computation and Competitive Analysis*. Cambridge University Press.
[6] Bin Cao, Chenyu Hou, Liwei Zhao, Louai Alarabi, Jing Fan, Mohamed F. Mokbel, and Anas Basalamah. 2020. SHAREK*: A scalable matching method for dynamic ride sharing. *GeoInformatica* 24, 4 (2020).
[7] Zhiguang Cao, Siwei Jiang, Jie Zhang, and Hongliang Guo. 2017. A unified framework for vehicle rerouting and traffic light control to reduce traffic congestion. *IEEE Trans. Intell. Transp. Syst.* 18, 7 (2017), 1958–1973.
[8] Moses Charikar and Balaji Raghavachari. 1998. The finite capacity dial-a-ride problem. In *FOCS*. 458–467.
[9] Lu Chen, Qilu Zhong, Xiaokui Xiao, Yunjun Gao, Pengfei Jin, and Christian S. Jensen. 2018. Price-and-time-aware dynamic ridesharing. In *ICDE*. 1061–1072.
[10] Peng Cheng, Hao Xin, and Lei Chen. 2017. Utility-aware ridesharing on road networks. In *SIGMOD*. 1197–1210.
[11] Didi Chuxing. 2018. *GAIA Initiative Open Dataset*. Retrieved July 22, 2018 from https://gaia.didichuxing.com/.
[12] Blerim Cici, Athina Markopoulou, and Nikolaos Laoutaris. 2015. Designing an on-line ride-sharing system. In *SIGSPATIAL*. 60:1–60:4.
[13] Michael J. Curry, John P. Dickerson, Karthik Abinav Sankararaman, Aravind Srinivasan, Yuhao Wan, and Pan Xu. 2019. Mix and match: Markov chains and mixing times for matching in rideshare. In *WINE*. 129–141.
[14] John P. Dickerson, Karthik Abinav Sankararaman, Aravind Srinivasan, and Pan Xu. 2018. Assigning tasks to workers based on historical data: Online task assignment with two-sided arrivals. In *AAMAS*. 318–326.
[15] John P. Dickerson, Karthik Abinav Sankararaman, Aravind Srinivasan, and Pan Xu. 2019. Balancing relevance and diversity in online bipartite matching via submodularity. In *AAAI*. 1877–1884.
[16] Pedro M. d'Orey, Ricardo Fernandes, and Michel Ferreira. 2012. Empirical evaluation of a dynamic and distributed taxi-sharing system. In *ITSC*. 140–146.
[17] R. Fagin, A. Lotem, and M. Naor. 2001. Optimal aggregation algorithms for middleware. In *PODS*. 102–113.
[18] E. Feuerstein and L. Stougie. 2001. On-line single-server dial-a-ride problems. *Theor. Comput. Sci.* 268, 1 (2001), 91–105.
[19] Luca Foti, Jane Lin, and Ouri Wolfson. 2021. Optimum versus Nash-equilibrium in taxi ridesharing. *GeoInformatica* 25, 3 (2021), 423–451.
[20] Xu Geng, Yaguang Li, Leye Wang, Lingyu Zhang, Qiang Yang, Jieping Ye, and Yan Liu. 2019. Spatiotemporal multi-graph convolution network for ride-hailing demand forecasting. In *AAAI*. 3656–3663.
[21] Geofabrik. 2021. *OpenStreetMap Data Extracts*. Retrieved July 18, 2021 from https://download.geofabrik.de/.
[22] Anupam Gupta, MohammadTaghi Hajiaghayi, Viswanath Nagarajan, and R. Ravi. 2010. Dial a ride from k-forest. *ACM Trans. Algorithms* 6, 2 (2010), 41.
[23] Wesam Mohamed Herbawi and Michael Weber. 2012. A genetic and insertion heuristic algorithm for solving the dynamic ridematching problem with time windows. In *GECCO*. 385–392.
[24] S. C. Ho, W. Y. Szeto, Y. H. Kuo, J. M. Y. Leung, M. Petering, and T. W. H. Tou. 2018. A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B: Methodological* 111 (2018).
[25] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. 2014. Large scale real-time ridesharing with service guarantee on road networks. *PVLDB* 7, 14 (2014), 2017–2028.
[26] J. J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. M. Wilson. 1986. A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. *Transportation Research Part B: Methodological* 20, 3 (1986), 243–257.
[27] Alexander Kleiner, Bernhard Nebel, and Vittorio A. Ziparo. 2011. A mechanism for dynamic ride sharing based on parallel auctions. In *IJCAI*. 266–272.
[28] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An experimental study on hub labeling based shortest path algorithms. *PVLDB* 11, 4 (2017), 445–457.
[29] Yafei Li, Ji Wan, Rui Chen, Jianliang Xu, Xiaoyi Fu, Hongyan Gu, Pei Lv, and Mingliang Xu. 2021. Top-k vehicle matching in social ridesharing: A price-aware approach. *IEEE Trans. Knowl. Data Eng.* 33, 3 (2021), 1251–1263.
[30] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. 2018. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. In *ICLR*.
[31] Z. Liu, Z. Gong, J. Li, and K. Wu. 2020. Mobility-aware dynamic taxi ridesharing. In *ICDE*. 961–972.
[32] Hui Luo, Zhifeng Bao, Farhana Murtaza Choudhury, and J. Shane Culpepper. 2021. Dynamic ridesharing in peak travel periods. *IEEE Trans. Knowl. Data Eng.* 33, 7 (2021), 2888–2902.
[33] Shuo Ma, Yu Zheng, and Ouri Wolfson. 2013. T-share: A large-scale dynamic taxi ridesharing service. In *ICDE*. 410–421.
[34] Shuo Ma, Yu Zheng, and Ouri Wolfson. 2015. Real-time city-scale taxi ridesharing. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1782–1795.
[35] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press.

[36] Vedant Nanda, Pan Xu, Karthik Abinav Sankararaman, John P. Dickerson, and Aravind Srinivasan. 2020. Balancing the tradeoff between profit and fairness in rideshare platforms during high-demand hours. In *AAAI*. 2210–2217.
[37] NYC. 2021. *TLC Trip Record Data*. Retrieved July 18, 2021 from https://www1.nyc.gov/site/tlc/index.page.
[38] Masayo Ota, Huy Vo, Claudio Silva, and Juliana Freire. 2015. A scalable approach for data-driven taxi ride-sharing simulation. In *Big Data*. 888–897.
[39] Masayo Ota, Huy Vo, Claudio Silva, and Juliana Freire. 2017. STaRS: Simulating taxi ride sharing at scale. *IEEE Trans. Big Data* 3, 3 (2017), 349–361.
[40] J. Pan, G. Li, and J. Hu. 2019. Ridesharing: Simulator, benchmark, and evaluation. *PVLDB* 12, 10 (2019), 1085–1098.
[41] Dominik Pelzer, Jiajian Xiao, Daniel Zehe, Michael H. Lees, Alois C. Knoll, and Heiko Aydt. 2015. A partition-based match making algorithm for dynamic ridesharing. *IEEE Trans. Intell. Transp. Syst.* 16, 5 (2015), 2587–2598.
[42] Zachary B. Rubinstein, Stephen F. Smith, and Laura Barbulescu. 2012. Incremental management of oversubscribed vehicle schedules in dynamic dial-a-ride problems. In *AAAI*. 1809–1815.
[43] Paolo Santi, Giovanni Resta, Michael Szell, Stanislav Sobolevsky, Steven H. Strogatz, and Carlo Ratti. 2014. Quantifying the benefits of vehicle pooling with shareability networks. *Proc. Natl. Acad. Sci.* 111, 37 (2014), 13290–13294.
[44] Douglas Oliveira Santos and Eduardo Candido Xavier. 2013. Dynamic taxi and ridesharing: A framework and heuristics for the optimization problem. In *IJCAI*. 2885–2891.
[45] Susan Shaheen, Adam Cohen, and Ismail Zohdy. 2016. *Shared Mobility: Current Practices and Guiding Principles*. Technical Report. United States. Federal Highway Administration.
[46] Bilong Shen, Yan Huang, and Ying Zhao. 2016. Dynamic ridesharing. *SIGSPATIAL Special* 7, 3 (2016), 3–10.
[47] R. S. Thangaraj, K. Mukherjee, G. Raravi, A. Metrewar, N. Annamaneni, and K. Chattopadhyay. 2017. Xhare-a-Ride: A search optimized dynamic ride sharing system with approximation guarantee. In *ICDE*. 1117–1128.
[48] Y. Tong, Y. Chen, Z. Zhou, L. Chen, J. Wang, Q. Yang, J. Ye, and W. Lv. 2017. The simpler the better: A unified approach to predicting original taxi demands based on large-scale online platforms. In *KDD*. 1653–1662.
[49] Yongxin Tong, Libin Wang, Zimu Zhou, Bolin Ding, Lei Chen, Jieping Ye, and Ke Xu. 2017. Flexible online task assignment in real-time spatial data. *PVLDB* 10, 11 (2017), 1334–1345.
[50] Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, Jieping Ye, and Ke Xu. 2018. A unified approach to route planning for shared mobility. *PVLDB* 11, 11 (2018), 1633–1646.
[51] Yongxin Tong, Zimu Zhou, Yuxiang Zeng, Lei Chen, and Cyrus Shahabi. 2020. Spatial crowdsourcing: A survey. *The VLDB Journal* 29, 1 (2020), 217–250.
[52] Paolo Toth and Daniele Vigo. 2002. *The Vehicle Routing Problem*. SIAM.
[53] Shared Use Mobility Center. 2021. *What is Shared Mobility?* Retrieved July 18, 2021 from https://goo.gl/3Jw6z7.
[54] Jiachuan Wang, Peng Cheng, Libin Zheng, Chao Feng, Lei Chen, Xuemin Lin, and Zheng Wang. 2020. Demand-aware route planning for shared mobility services. *PVLDB* 13, 7 (2020), 979–991.
[55] David Wilkie, Cenk Baykal, and Ming C. Lin. 2014. Participatory route planning. In *SIGSPATIAL*. 213–222.
[56] D. Wilkie, J. P. Berg, M. C. Lin, and D. Manocha. 2011. Self-aware traffic route planning. In *AAAI*.
[57] N. H. M. Wilson, R. Weissberg, B. Higonnet, and J. Hauser. 1975. *Advanced Dial-A-Ride Algorithms*. Technical Report.
[58] Yi Xu, Yongxin Tong, Yexuan Shi, Qian Tao, Ke Xu, and Wei Li. 2019. An efficient insertion operator in dynamic ridesharing services. In *ICDE*. 1022–1033.
[59] Y. Xu and P. Xu. 2020. Trade the system efficiency for the income equality of drivers in rideshare. In *IJCAI*. 4199–4205.
[60] Yifan Xu, Pan Xu, Jianping Pan, and Jun Tao. 2020. A unified model for the two-stage offline-then-online resource allocation. In *IJCAI*. 4206–4212.
[61] Andrew Chi Chin Yao. 1977. Probabilistic computations: Toward a unified measure of complexity. In *FOCS*. 222–227.
[62] Huaxiu Yao, Fei Wu, Jintao Ke, Xianfeng Tang, Yitian Jia, Siyu Lu, Pinghua Gong, Jieping Ye, and Zhenhui Li. 2018. Deep multi-view spatial-temporal network for taxi demand prediction. In *AAAI*. 2588–2595.
[63] San Yeung, Evan Miller, and Sanjay Madria. 2016. A flexible real-time ridesharing system considering current road conditions. In *MDM*. 186–191.
[64] Chak Fai Yuen, Abhishek Pratap Singh, Sagar Goyal, Sayan Ranu, and Amitabha Bagchi. 2019. Beyond shortest paths: Route recommendations for ride-sharing. In *WWW*. 2258–2269.
[65] Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2019. Last-mile delivery made practical: An efficient route planning framework with theoretical guarantees. *PVLDB* 13, 3 (2019), 320–333.
[66] Yuxiang Zeng, Yongxin Tong, Yuguang Song, and Lei Chen. 2020. The simpler the better: An indexing approach for shared-route planning queries. *PVLDB* 13, 13 (2020), 3517–3530.
[67] Junbo Zhang, Yu Zheng, and Dekang Qi. 2017. Deep spatio-temporal residual networks for citywide crowd flows prediction. In *AAAI*. 1655–1661.
[68] Boming Zhao, Pan Xu, Yexuan Shi, Yongxin Tong, Zimu Zhou, and Yuxiang Zeng. 2019. Preference-aware task assignment in on-demand taxi dispatching: An online stable matching approach. In *AAAI*. 2245–2252.

[69] Bolong Zheng, Chenze Huang, Christian S. Jensen, Lu Chen, Nguyen Quoc Viet Hung, Guanfeng Liu, Guohui Li, and Kai Zheng. 2020. Online trichromatic pickup and delivery scheduling in spatial crowdsourcing. In *ICDE*. 973–984.
[70] Libin Zheng, Lei Chen, and Jieping Ye. 2018. Order dispatch in price-aware ridesharing. *PVLDB* 11, 8 (2018), 853–865.
[71] L. Zheng, P. Cheng, and L. Chen. 2019. Auction-based order dispatch and pricing in ridesharing. In *ICDE*. 1034–1045.

Received January 2021; revised July 2021; accepted September 2021