A Unified Approach to Route Planning for Shared Mobility

Yongxin Tong † Yuxiang Zeng ‡ Zimu Zhou # Lei Chen ‡ Jieping Ye § Ke Xu †
† SKLSDE Lab, BDBC, and IRI, Beihang University, China
‡ The Hong Kong University of Science and Technology, Hong Kong SAR, China
# ETH Zurich, Zurich, Switzerland, §Didi Research Institute, Didi Chuxing, Beijing, China
†{yxtong,kexu}@buaa.edu.cn, ‡{yzengal,leichen}@cse.ust.hk,
#zzhou@tik.ee.ethz.ch, §yejieping@didichuxing.com

ABSTRACT

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 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 prove the problem is NP-hard and there is no polynomial-time algorithm with constant competitive ratio for the URPSM problem and its variants. In response, we devise an effective and efficient solution to address the URPSM problem approximately. We design a novel dynamic programming (DP) algorithm to accelerate the insertion operation from cubic or quadric time in previous work to only linear time. On basis of the DP algorithm, we propose a greedy based solution to the URPSM problem. Experimental results on real datasets show that our solution outperforms the state-of-the-arts by 1.2 to 12.8 times in effectiveness, and also runs 2.6 to 20.7 times faster.

PVLDB Reference Format:
Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, Jieping Ye, Ke Xu. A Unified Approach to Route Planning for Shared Mobility. *PVLDB*, 11(11): xxxx-yyyy, 2018.
DOI: https://doi.org/10.14778/3236187.3236211

1. INTRODUCTION

Shared mobility refers to transportation services that are shared among users, such as ride-sharing, food delivery and crowdsourced parcel delivery [40]. By altering routes and filling under-used vehicles, shared mobility mitigates pollution, reduces transportation costs, and provides last-mile delivery solutions [41]. 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 [30][25][34][42][39][24]; *(ii)* maximizing the number of served requests [19][47][21][29][39][24]; and *(iii)* maximizing the total revenue [13][14]. 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 [30][25][42][34][19][47][39][18]. In practice, previous studies have the following limitations.

Limitation 1. Existing proposals sometimes adopt multiple vague or even conflicted optimization objectives. For example, in [30][25][34][39][24], the goal is to minimize the total travel distance of requests without specifying how many requests should be served. Hence an "optimal" solution is to serve no request at all, which contradicts to 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 various real-world shared mobility applications.

Limitation 2. The *insertion* operation in existing solutions [30][25][47][18][31] are inefficient for large-scale shared mobility platforms. It takes at least square time to insert a new request into a route, making insertion a bottleneck to process large numbers of requests in real-world applications.

To address these limitations, we define a new problem, 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, we design a novel dynamic

Proceedings of the VLDB Endowment, Vol. 11, No. 11

DOI: https://doi.org/10.14778/3236187.3236211

programming (DP) algorithm that reduces its time complexity from cubic or quadric [18][30][25][19][47] to linear. The key insight is that dynamic programming can be utilized to find a best pickup location in $O(1)$ time.

Furthermore, 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 constant competitive ratio for the URPSM problem and its special cases studied in previous literature [30][25][18]. We finally devise an effective and efficient heuristic solution to the URPSM problem.

Our main contributions can be 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 insertion operation. Our algorithm reduces the time complexity of this basic operation from cubic or quadric to linear.
- We comprehensively analyze the hardness of approximation of the URPSM problem. Specifically, we prove that there is no polynomial-time algorithm with constant competitive ratio for the URPSM problem and its variants. The results serve as references to analyze other route planning problems and guidelines to design efficient solutions to the URPSM problem.
- We devise an effective and efficient solution using the DP-based insertion to solve the URPSM problem.
- Extensive experiments on real datasets show that our solution is 2.6 to 20.7 times faster and 1.2 to 12.8 times more effective than the state-of-the-arts [25][11].

In the rest of this paper, we review related work in Sec. 2, formulate the URPSM problem, and discuss its generalizability as well as its hardness in Sec. 3. We propose a dynamic programming based insertion in Sec. 4 and design a complete solution to the URPSM problem in Sec. 5. Finally we present the evaluations in Sec. 6 and conclude in Sec. 7.

2. RELATED WORK

Research on route planning for shared mobility (RPSM) dates back to the dial-a-ride problem proposed in 1975 [44][45], and has been studied by the database, data mining, transportation science communities. This section briefly reviews different variants of the RPSM problem and their solutions.

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 [30][25][42][34][47][19][13][14] and will be our main focus.

Mainstream objectives of RPSM problems include minimizing the total travel distance [16][23], maximizing the number of served requests [29][47][19][39], maximizing the total revenue [13][14], etc. The total travel distance calculates the total distance traveled by the workers to serve the requests. A small total travel distance indicates a low travel cost and little pollution [10]. A large number of served requests contribute to the revenue of the shared mobility providers [47]. A more common goal is to minimize the total travel distance while serving all the requests [30][25][42][34]. Other studies focus on maximizing the total revenue of the shared mobility provider (the total payment of the served requests minus the total salaries of the workers) [13][14], minimizing the makespan (the completion time of the last request) [12][22], or maximizing the complicated social utilities between workers and requests [18]. Our aim is to analyze the relationship among mainstream objectives and integrate them into a compatible and flexible formulation.

Many solutions to the dynamic RPSM problems have been proposed [30][25][42][34][47][19], where a core operation, called *insertion*, is widely utilized. Zheng *et al.* [30][42] use the enumeration strategy to search 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 [34][37]. Parallelism also applies to speed up insertion [34]. Insertion is frequently used in the solutions to large-scale dynamic RPSM problems. However, the insertion has quadric or even cubic time complexity, which is a bottleneck of efficiency. This motivates us to devise a linear insertion algorithm.

To solve the dynamic RPSM problems, Zheng *et al.* [30][42] first search a set of candidate workers through grid index and then insert the request to the candidate with minimal increased distance. Huang *et al.* [25] propose a kinetic data structure to store all possible routes and use a similar insertion procedure to minimize the total travel distance. Alonso-Mora *et al.* [11] adopt a batch-based method to first divide a few requests into small groups, and then insert a group of requests into the route of one worker. However, these studies are unfit for large-scale shared mobility applications. On basis of a novel linear insertion, we propose a complete heuristic solution to the RPSM problem, which is both more effective and efficient than these studies.

3. PROBLEM STATEMENT

This section defines the URPSM problem, which unifies the objective functions of many prior studies [30][25][34][42][47][21][29][39][24][13][14].

3.1 Notations and Definitions

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 cost $cost(u, v)$.*

The travel cost can be either a distance or an average travel time, which can be obtained from OpenStreetMap [6] or large historical trajectory mining [48]. We use travel time and travel distance interchangeably in this paper. We denote $dis(u, v)$ as the distance of the shortest path between any two vertices $u \in V$ and $v \in V$.

DEFINITION 2 (WORKER). *A worker is denoted by $w =< o_w, K_w >$ with an initial location $o_w \in V$ and a capacity K_w.*

The capacity of a worker 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. We use $W = \{w_1 \cdots w_{|W|}\}$ to denote all the workers.

Definition 3 (Request). *A request is denoted by* $r = < o_r, d_r, t_r, e_r, p_r, K_r >$ *with an origin* $o_r \in V$*, a destination* $d_r \in V$*, and a capacity* K_r*. It is released on the shared mobility platform (platform for short) at a release time* t_r *and needs to be served before a deadline* e_r*. A request is served if (i) a worker picks up* r *at* o_r *after* t_r*; and (ii) the same worker drops* r *at* d_r *before* e_r*. If a request is not served (rejected), the platform will receive a penalty* p_r*.*

The capacity K_r of a request specifies the number of passengers in ride-sharing or items in courier services in a single request. Note that there can be two deadlines in real-world applications, *i.e.*, the deadlines for *pickup* and *delivery*. Yet a single deadline for delivery e_r usually suffices since the deadline for pickup can be expressed as $e_r - dis(o_r, d_r)$. Note that it is difficult to serve every request given a tight deadline (*e.g.*, 5-6 minutes in ride-sharing [30][13]). Hence a platform may reject 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 $R = \{r_1 \cdots r_{|R|}\}$ to denote all the requests and R_w to denote all the requests served by worker w. We further denote $R^+ = \bigcup_{w \in W} R_w$ 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* $S_w = \langle o_w, l_w^1, \cdots, l_w^{|S_w|-1} \rangle$*, where* $\langle l_w^1, \cdots, l_w^{|S_w|-1} \rangle$ *is an ordered sequence of origin and destination of* R_w, i.e., $l_w^i \in \{o_r \mid r \in R_w\} \cup \{d_r \mid r \in R_w\}$*. A route is feasible if (i)* $\forall r \in R_w$*,* o_r *and* d_r *exist and* o_r *precedes* d_r *in the sequence; (ii)* $\forall r \in R_w$*, the time when* w *arrives at* d_r *is no later than the deadline* e_r*; (iii) At any time, the number of passengers/items that have been picked up but not delivered in this route, does not exceed the capacity of the worker.*

We use $D(S_w)$ to denote the total travel distance of S_w, *i.e.*, $D(S_w) = dis(o_w, l_w^1) + \sum_{i=2}^{|S_w|-1} dis(l_w^{i-1}, l_w^i)$.

3.2 Unified Objective and URPSM Problem

Definition 5 (URPSM). *Given a road network, a set of workers* W*, a set of requests 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 the following constraints: (i) Feasibility constraint: each worker is arranged a feasible route; (ii) Invariable constraint: once requests are rejected, they cannot be revoked. Otherwise, they must be served.

We illustrate the URPSM problem by the following example.

Example 1. *Suppose a ride-sharing platform with two workers (vehicles)* w_1, w_2 *and three dynamically arrived requests* r_1*-*r_3*. The initial locations of workers are labeled on a road network with eight vertices* v_1*-*v_8 *as shown in Fig. 1. The coordinates (latitudes and longitudes) of the vertices are also labeled. For example, the coordinate of* v_1 *is* $(0, 1)$*. Table 1 lists the details of the requests. We assume* $\alpha = 1$*,* $K_{w_1} = K_{w_2} = 4$ *and* $K_{r_1} = K_{r_2} = K_{r_3} = 1$*.*

At time 0 (t_{r_1}*), a request* r_1 *is released with 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 23. A feasible route is* $\langle o_{w_1}, v_2, v_4 \rangle$*, which reaches* v_4 *at time* $5 + 1 + 5 + 5 = 16$*. Specifically,* w_1 *starts from* v_7 *and first travels from* v_1 *to* v_2*.* w_1 *picks up* r_1 *at* v_2 *and then travels from* v_8 *to* v_4*. Finally* w_1 *takes* r_1 *to the destination before the deadline* $e_{r_1} = 23$*. The platform can also reject the request, which will incur a penalty* $p_{r_1} = 20$*. The URPSM problem plans routes for each worker and minimize the unified cost, which is composed of both the total travel distance and the penalty of unserved requests.*

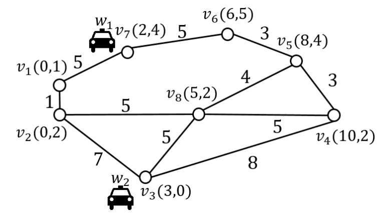


Figure 1: A road network with coordinates.

Table 1: Dynamically arrived requests.

request	release time (t_r)	deadline (e_r)	origin (o_r)	destination (d_r)	penalty (p_r)
r_1	0	23	v_2	v_4	20
r_2	5	26	v_3	v_5	10
r_3	11	25	v_8	v_5	9

Next we show that many previous studies are special cases of our URPSM problem with specific α and p_r settings.

- *Minimize the total travel distance* [25][42][30][33][35]. By setting $\alpha = 1$ and $\forall r \in R, p_r = \infty$, minimizing Eq. (1) is equivalent to minimizing the total travel distance while serving all requests.
- *Maximize the number of served requests* [47][19][29][21]. By setting $\alpha = 0$ and $\forall r \in R, p_r = 1$, minimizing Eq. (1) is equivalent to minimizing the number of unserved requests (*i.e.*, maximizing the number of served requests) since the penalty of any r is $p_r = 1$.
- *Maximize the total revenue* [13][14]. 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 (or travel distance) and the income for unit time c_w. The fare of a request is relevant to the travel distance and the fare for unit distance c_r. Then the total revenue of the platform is calculated as:

$$revenue(W, R) = c_r \sum_{r \in R^+} dis(o_r, d_r) - c_w \sum_{w \in W} D(S_w) \tag{2}$$

Set $\alpha = c_w$ and $\forall r \in R, p_r = c_r \times dis(o_r, d_r)$:

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

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

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

Since the requests are given (*i.e.*, the first term is a constant), minimizing $UC(W, R)$ is equivalent to maximizing the total revenue.

We summarize the major notations in Table 2.

Table 2: Summary of major notations.

Notation	Description
R, W	a set of requesters and workers
o_r, d_r	origin and destination of request r
p_r	penalty of each unserved request r
R^+, R^-	a set of served requests and rejected requests
$S_w, D(S_w)$	schedule of worker w and its distance
K_w, K_r	capacity of worker w, capacity of request r
α	weight parameter for unit distance of workers
$dis(.,.)$	shortest distance between two vertices

3.3 Hardness Analysis

This subsection analyzes the *competitive hardness* of the URPSM problem and its variants. The URPSM problem is NP-hard since it generalizes the existing NP-hard problems [30]. However, there are few studies on the *competitive hardness*. The only known result [13] proves that no deterministic algorithm can guarantee constant competitive ratio to maximize the total revenue, but it is unknown whether the conclusion applies to randomized algorithms. We analyze the competitive hardness by studying whether a randomized algorithm can guarantee constant competitive ratio against an oblivious adversary [15]. If no such randomized algorithm exists, nor will any deterministic algorithm [15].

Theorem 1 presents our main results.

THEOREM 1. *The following special cases of the URPSM problem has no constant competitive ratio for either randomized or deterministic algorithms:*
(1) maximizing the number of served requests, i.e., $\alpha = 0$ *and* $\forall r \in R, p_r = 1$*;*
(2) maximizing the total revenue of the platform, i.e., $\alpha = c_w$ *and* $\forall r \in R, p_r = c_r \times dis(o_r, d_r)$*;*
(3) minimizing the total distance while serving all requests, i.e., $\alpha = 1$ *and* $p_r = \infty$.

We prove the three statements in Theorem 1 sequentially by Lemma 1, Lemma 2 and Lemma 3, respectively.

LEMMA 1. *When* $\alpha = 0$ *and* $\forall r \in R, p_r = 1$*, neither a randomized nor a deterministic algorithm has a constant competitive ratio.*

PROOF. We only need to show that no randomized algorithm can guarantee 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 [46], no randomized algorithm has a constant competitive ratio.

The distribution χ of the requests, workers and road network is generated as follows: *(i)* 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. *(ii)* We assume a single worker with initial location $o_w = v_1$ and capacity $K_w = 2$. *(iii)* 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 $p_r = K_r = 1$.

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.*, $E_\chi[OPT] = 0$.

Consider a generic deterministic online algorithm ALG 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, ALG is able to serve r with a probability $\le \frac{2\epsilon}{|V|}$. Since there is only one request, the expected number of unserved requests of ALG is $E_\chi[ALG] \ge 1 - \frac{2\epsilon}{|V|}$. Hence

$$\frac{E_\chi[ALG]}{E_\chi[OPT]} \ge \frac{1 - \frac{2\epsilon}{|V|}}{0} \to \infty$$

The above ratio becomes unbounded. □

LEMMA 2. *When* $\alpha = c_w$ *and* $\forall r \in R, p_r = c_r \times dis(o_r, d_r)$*, neither a randomized nor a deterministic algorithm has a constant competitive ratio.*

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 $dis(o_r, d_r) = |V|/2$, 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 c_r *e.g.*, $c_r > 2c_w$, otherwise an optimal solution may reject r when the total distance of the worker is close to $|V|$. Then we have

$$E_\chi[OPT] \le \alpha|V| = c_w|V|$$
$$E_\chi[ALG] \ge (1 - \frac{2\epsilon}{|V|}) \cdot p_r = (1 - \frac{2\epsilon}{|V|}) \cdot c_r \cdot \frac{|V|}{2}$$

If ϵ is small enough, then

$$\frac{E_\chi[ALG]}{E_\chi[OPT]} \ge \frac{c_r}{2c_w}(1 - \frac{2\epsilon}{|V|}) = \Omega(\frac{c_r}{c_w})$$

Therefore neither a randomized nor a deterministic algorithm has a constant competitive ratio. □

LEMMA 3. *When* $\alpha = 1$ *and* $p_r = \infty$*, neither a randomized nor a deterministic algorithm has a constant competitive ratio.*

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,

$$E_\chi[OPT] \le \alpha|V| = |V|, \; E_\chi[ALG] \ge (1 - \frac{2\epsilon}{|V|}) \cdot p_r$$
$$\frac{E_\chi[ALG]}{E_\chi[OPT]} \ge \frac{p_r}{|V|}(1 - \frac{2\epsilon}{|V|})$$

By setting a sufficiently small ϵ and $p_r = \infty$, the above ratio becomes unbounded, *i.e.*, $\frac{E_\chi[ALG]}{E_\chi[OPT]} \to \infty$. □

4. DP-BASED INSERTION

Although there is no algorithm with provable effectiveness to solve the URPSM problem (Sec. 3.3), solutions built upon *insertion* prove to be practically effective for the variants of the URPSM problem [30][25][42][34]. However, the insertion operation is also an efficiency bottleneck in large-scale dynamic shared mobility applications. This section formally defines the insertion operation, and proposes a novel DP-based algorithm to boost its efficiency.

4.1 Preliminaries

Insertion was first proposed in [32] for the *Vehicle Routing Problem* (VRP) [20], which arranges optimal routes for a set of vehicles to deliver a given set of requests (passengers) to

Algorithm 1: Basic Insertion

input : a worker w with route S_w and a request r
output: a new route S^* for the worker w
1 $S^* \leftarrow S_w, \Delta^* \leftarrow \infty$;
2 **foreach** $i \leftarrow 0$ **to** n **do**
3   **foreach** $j \leftarrow i$ **to** n **do**
4         $S'_w \leftarrow$ insert o_r at i-th and d_r at j-th in S_w;
5   **if** S'_w *is feasible* **then**
6             $\Delta_{i,j} \leftarrow D(S'_w) - D(S_w)$;
7   **if** $\Delta_{i,j} < \Delta^*$ **then** $S^* \leftarrow S'_w, \Delta^* \leftarrow \Delta_{i,j}$;
8 **return** S^*;

different cities. The idea of an insertion-based solution is to iteratively arrange a route for a vehicle by inserting one vertex (city) at a time. This idea can be extended by inserting two vertices (*i.e.*, origin and destination of the request) at a time, and has been used to design heuristic solutions to the dial-a-ride problem and its variants [27][28][36][26]. Although the insertion is proposed for route planning for a single worker, it has also been widely adopted in multi-worker route planning, where the insertion-based route planning is performed for each worker individually [18][30][25].

Formally, we define the insertion operation following the conventions in [27][28] as follows.

Definition 6 (Insertion). *Given a worker w with the current route S_w composed of n vertices, and a new request r, the insertion operation aims to find a new feasible route S^* with the minimal increased distance to further serve r, by inserting both o_r and d_r into S_w, such that the order of vertices in S_w remains the same in S^*.*

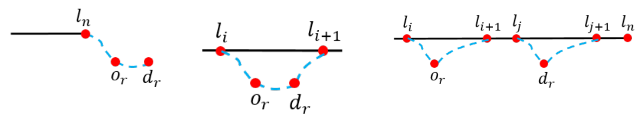
By inserting a pair of origin and destination that has a minimal increased distance, it also minimizes the total travel distance. Thus the goal is aligned with our URPSM problem, which minimizes the weighted total distance and the penalty of unserved requests. We first review the basic insertion proposed in previous studies [27], then design a naive DP-based insertion with quadric time complexity and linear memory complexity, and an improved version with both linear time and memory complexities.

4.2 Basic Insertion

Basic insertion was proposed in [27][28] without optimization for efficiency. Its idea 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 *(3)* replace S^* by S'_w if no constraint is violated and S'_w increases a shorter distance.

Algo. 1 illustrates the basic insertion. In line 1, it initializes a new route S^* as S_w in case of no feasible route. In lines 2-3, we enumerate all possible pickup places (at i-th in S_w) and deliver places (at j-th in S_w). In lines 4-7, we generate a new route S'_w and check whether it is feasible. If yes, we calculate its increased distance $\Delta_{i,j}$ and compare it to the current minimal Δ^*. We update S^* using S'_w and Δ^* using $\Delta_{i,j}$ if $\Delta_{i,j} < \Delta^*$.

Complexity Analysis. The number of possible (i, j) pairs is $O(n^2)$ in lines 2-3. Line 5 checks whether the new route S'_w violates the capacity and the deadline constraints in $O(n)$ time. If not, it will take $O(n)$ time to calculate the increased distance in line 6. Note that lines 5-6 involve shortest distance queries. The above analysis assumes a shortest distance query takes $O(1)$ time, as with many previous studies [11][13][18][33]. The assumption is reasonable because real-world shared mobility companies like Didi [3] regard shortest distance queries as a basic operation of constant time. We will adopt this assumption throughout the rest of this paper unless explicitly specified. Lines 5-7 check S'_w and update S^* and Δ^*, which also takes $O(n)$ time. Thus the total time cost of basic insertion is $O(n^3)$ and the total memory cost is $O(n)$. If a shortest distance query takes $O(q)$ time, the algorithm takes $O(n^3 q)$ time.



(a) $i = j = n$ (b) $i = j < n$ (c) $i < j$

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

4.3 Naive DP-Based Insertion

This subsection presents a naive DP-based insertion algorithm, which reduces the $O(n^3)$ time complexity of basic insertion to $O(n^2)$. The idea is to *(1)* enumerate all possible pairs of places for inserting o_r and d_r, but *(2)* check whether a new route is feasible and calculate $\Delta_{i,j}$ in $O(1)$ time instead of $O(n)$ time in basic insertion. We first explain how to calculate $\Delta_{i,j}$ because it will be used to check the feasibility of a new route.

4.3.1 *Calculating $\Delta_{i,j}$ in $O(1)$ Time*

Rather than calculate $\Delta_{i,j} = D(S'_w) - D(S_w)$ from scratch, which takes $O(n)$ time, we calculate $\Delta_{i,j}$ in $O(1)$ time leveraging the *detour* when inserting l_j between l_i and l_k. Specifically, the detour $det(l_i, l_j, l_k)$ is defined as follows.

$$det(l_i, l_j, l_k) = dis(l_i, l_j) + dis(l_j, l_k) - dis(l_i, l_k)$$

It is the increased distance when l_j is inserted between l_i and l_k. Accordingly, $\Delta_{i,j}$ can be calculated in $O(1)$ time using detours in Fig. 2, where Fig. 2a and Fig. 2b are two special cases when $i = j$ while Fig. 2c is the general case.

$$\Delta_{i,j} = \begin{cases} dis(l_n, o_r) + dis(o_r, d_r), & \text{if } i = j = n \\ dis(l_i, o_r) + dis(o_r, d_r) & \\ \quad +dis(d_r, l_{i+1}) - dis(l_i, l_{i+1}), & \text{if } i = j < n \\ det(l_i, o_r, l_{i+1}) + det(l_j, d_r, l_{j+1}), & \text{otherwise} \end{cases} \tag{5}$$

4.3.2 *Checking Route Feasibility in $O(1)$ Time*

To check whether a new route is feasible, we should check *(i)* the deadline constraint and *(ii)* the capacity constraint defined in Definition 4.

To check the deadline constraint in $O(1)$ time, we borrow the idea of slack time [27]. Denote $ddl[k]$ as the latest time to arrive at l_k without violating any deadline constraint. We set $ddl[k]$ as $e_r - dis(o_r, d_r)$ if l_k is the origin of r; and e_r if l_k is the destination.

$$ddl[k] = \begin{cases} e_r - dis(o_r, d_r), & \text{if } l_k \text{ is } o_r \\ e_r, & \text{if } l_k \text{ is } d_r \end{cases} \tag{6}$$

Denote $arr[k]$ as the time when w arrives at l_k. Hence,

$$arr[k] = arr[k-1] + dis(l_{k-1}, l_k) \tag{7}$$

Further denote $slack[k]$ as the maximal tolerable time for *detour* (*i.e.*, slack time) between l_k and l_{k+1} to satisfy all

Algorithm 2: Naive DP Insertion

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

1 $S^* \leftarrow S_w, \Delta^* \leftarrow \infty$;
2 Initialize $ddl, arr, slack, picked$ by Eq. (6)-Eq. (9);
3 **foreach** $i \leftarrow 0$ **to** n **do**
4   **if** *(1) of Lemma 4 is violated* **then** break;
5   **if** *(1) of Lemma 5 is violated* **then** continue;
6   **if** *(2) of Lemma 4 is violated* **then** continue;
7   **foreach** $j \leftarrow i$ **to** n **do**
8   **if** *$j > i$ and $picked[j] > K_w - K_r$* (i.e., *(2) of Lemma 5 is violated)* **then** break;
9   calculate $\Delta_{i,j}$ by Eq. (5);
10   **if** *(3) or (4) in Lemma 4 is violated* **then** break;
11   **if** $\Delta_{i,j} < \Delta^*$ **then** $\Delta^* \leftarrow \Delta_{i,j}, i^* \leftarrow i, j^* \leftarrow j$;
12 **if** $\Delta^* < \infty$ **then**
    $S^* \leftarrow$ insert o_r at i^*-th and d_r at j^*-th in S_w;
13 **return** S^*;

the deadlines after l_k. Since a *detour* after l_k may cause the worker to violate the deadlines of $l_{k+1}, \cdots, l_n$, $slack[k]$ should be no larger than the deadline of location l_{k+1} ($ddl[k+1] - arr[k+1]$), and all the deadlines after location l_{k+1} ($slack[k+1]$). Hence, $slack[\cdot]$ can be calculated as follows.

$$\begin{aligned} slack[k] &= \min_{k'>k} (ddl[k'] - arr[k']) \\ &= \min\{slack[k+1], ddl[k+1] - arr[k+1]\} \end{aligned} \quad (8)$$

LEMMA 4. *The deadline constraint will not be violated if and only if (1) $arr[i] + dis(l_i, o_r) \le e_r$; (2) $det(l_i, o_r, l_{i+1}) \le slack[i]$; (3) $arr[i] + dis(l_i, o_r) + dis(o_r, d_r) \le e_r$ when $i = j$ (Fig. 2a and Fig. 2b) or $arr[j] + det(l_i, o_r, l_{i+1}) + dis(l_j, d_r) \le e_r$ when $i < j$ (Fig. 2c); and (4) $\Delta_{i,j} \le slack[j]$.*

PROOF (SKETCH). Condition (1) checks whether the deadline constraint of the new request r is violated if o_r is inserted at i-th; condition (2) checks whether any deadline constraint of all the other requests is violated if o_r is inserted at i-th. Similarly, condition (3) checks whether the deadline constraint of r is violated if d_r is inserted at j-th; condition (4) checks whether any deadline constraint of all the other requests is violated if d_r is inserted at j-th. We refer readers to [43] for more details. □

To check the capacity constraint in $O(1)$ time, we use *picked request* (*i.e.*, $picked[k]$) to denote the number of requests currently picked up yet not delivered. Then we have

$$picked[k] = \begin{cases} picked[k-1] + K_r, & \text{if } l_k \text{ is } o_r \\ picked[k-1] - K_r, & \text{if } l_k \text{ is } d_r \end{cases} \quad (9)$$

LEMMA 5. *The capacity constraint of worker w will not be violated if and only if (1) $picked[i] \le K_w - K_r$, and (2) $\forall k, i < k \le j, picked[k] \le K_w - K_r$.*

PROOF. Since o_r is inserted at i-th, $picked[i]$ should be no greater than $K_w - K_r$. We also need to check whether there exists any integer $k \in (i, j]$ such that $picked[k]$ is greater than $K_w - K_r$. As the request r is delivered right after l_j, this will exceed the worker capacity K_w at l_k if $picked[k] > K_w - K_r$, which violates the capacity constraint of w. □

4.3.3 Algorithm Sketch

Algo. 2 shows the naive DP insertion with initialization in line 2 by Eq. (6)-Eq. (9). Line 9 is calculated by Eq. (5). Line 4, 6 and 10 implement Lemma 4. Line 5 and 8 implement Lemma 5.

Complexity Analysis. Line 2 takes $O(n)$ time to initialize $arr, ddl, slack, picked$. Line 12 takes $O(n)$ time and there are still $O(n^2)$ iterations in lines 3-11. The other lines all take $O(1)$ time. The total time cost is $O(n^2)$ and the total memory cost is $O(n)$. If a shortest distance query takes $O(q)$ time, the algorithm takes $O(n^2 q)$ time.

4.4 Linear DP-Based Insertion

This subsection presents an improved DP-based insertion with linear time complexity (linear DP insertion for short). It finds the route with the minimal increased distance without enumerating all possible pairs of places (i, j) for insertion. Linear DP insertion is built upon the naive DP insertion, but leverages two insights. *(i)* It only takes $O(n)$ time to find the best route for the special cases when $i = j$ as in Fig. 2a and Fig. 2b. *(ii)* Given a fixed j, it only takes $O(1)$ time to find the best i via dynamic programming in the general case as in Fig. 2c. The first insight is trivial because it takes $O(1)$ time to check the feasibility of S'_w and calculate Δ^* (Sec. 4.3). Thus when $i = j$, it takes $O(n)$ time to find the feasible route with the minimal increased distance. In the following, we mainly explain the second insight.

4.4.1 Enumerating Delivery Locations

Instead of enumerating all possible pairs (i, j), linear DP insertion only enumerates the delivery locations (j) to find the best route. Denote Δ_j^* as the minimal increased distance for a given j. For the general case in Fig. 2c,

$$\begin{aligned} \Delta_j^* &= \min_{i<j} \Delta_{i,j} \\ &= \min_{i<j} \big(det(l_i, o_r, l_{i+1}) + det(l_j, d_r, l_{j+1})\big) \\ &= det(l_j, d_r, l_{j+1}) + \min_{i<j} det(l_i, o_r, l_{i+1}) \end{aligned} \quad (10)$$

The first term is the detour after inserting j, which is constant for a fixed j. The second term is the minimal detour among all $i < j$. The key in linear DP insertion is to find a feasible i to minimize the second term in $O(1)$ time.

4.4.2 Finding the Best Pickup Location in $O(1)$ Time

We introduce $Dio[j]$ to maintain the minimal detour for inserting o_r among $i < j$ for a given j. That is, $Dio[j] = \min_{i<j} det(l_i, o_r, l_{i+1})$. $Dio[j]$ can be calculated via the following DP formulation.

$$Dio[j] = \begin{cases} \infty, \text{ if } picked[j-1] > K_w - K_r \\ Dio[j-1], \text{ if } det(l_{j-1}, o_r, l_j) > slack[j-1] \\ \min\{Dio[j-1], det(l_{j-1}, o_r, l_j)\}, \text{ otherwise} \end{cases} \quad (11)$$

The first case comes from Lemma 5, the second case comes from Lemma 4, and the third case is due to its definition.

Similarly, we use $Plc[j]$ to record the insertion place of o_r corresponding to $Dio[j]$.

$$Plc[j] = \begin{cases} NIL, & \text{if } picked[j-1] > K_w - K_r \\ Plc[j-1], & \text{if } det(l_{j-1}, o_r, l_j) > slack[j-1] \\ Plc[j-1], & \text{if } Dio[j-1] < det(l_{j-1}, o_r, l_j) \\ j-1, & \text{if } Dio[j-1] \ge det(l_{j-1}, o_r, l_j) \end{cases} \quad (12)$$

If $Plc[j]$ and j satisfy the capacity and deadline constraints, then we obtain the best feasible route for a fixed j. However,

Algorithm 3: Linear DP Insertion

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

1 $S^* \leftarrow S_w, \Delta^* \leftarrow \infty, Dio[0] \leftarrow \infty, Plc[0] \leftarrow NIL$;
2 Initialize $ddl, arr, slack, picked$ by Eq. (6)-Eq. (9);
3 **foreach** $j \leftarrow 0$ **to** n **do**
4   Update Δ^*, i^*, j^* with special cases as shown in Fig. 2a and Fig. 2b;
5   **if** $j > 0$ *and Corollary 1 is satisfiable* **then**
6         $\Delta_j^* \leftarrow det(l_j, d_r, l_{j+1}) + Dio[j]$;
7   **if** $\Delta_j^* < \Delta^*$ **then**
            $\Delta^* \leftarrow \Delta_j^*, i^* \leftarrow Plc[j], j^* \leftarrow j$;
8   **if** $arr[j] + dis(o_r, e_r) > e_r$ **then** break;
9   Update $Dio[j+1]$ and $Plc[j+1]$ according to Eq. (11) and Eq. (12);
10 **if** $\Delta^* < \infty$ **then**
    $S^* \leftarrow$ insert o_r at i^*-th and d_r at j^*-th in S_w;
11 **return** S^*;

if $Plc[j]$ violates certain constraint, it is unknown whether there is certain $i \neq Plc[j]$ that may generate a feasible route. We tackle this problem via the following lemma.

Lemma 6. *If $Plc[j]$ violates the constraints, then other $i \neq Plc[j]$ will also violate the constraints.*

Proof. First, assume $Plc[j]$ violates the capacity constraint (the first condition of Eq. (12)). According to Lemma 5, any $i \leq j-1$ will also violate the capacity constraint. Next, assume $Plc[j]$ violates the deadline constraint (the second condition of Eq. (12)). Suppose to the contrary, there exists $i' < j$ which satisfies all constraints. Then we have

$$det(l_{i'}, o_r, l_{i'+1}) + det(l_j, d_r, l_{j+1}) \leq slack[j] \quad (13)$$

$$det(l_{i'}, o_r, l_{i'+1}) + det(l_j, d_r, l_{j+1}) \leq e_r - arr[j] \quad (14)$$

According to Eq. (11), $Plc[j]$ can only violate the deadline constraint together with the given j, *i.e.*, $det(l_{i'}, o_r, l_{i'+1}) \leq Dio[j]$. It indicates that $Plc[j]$ should also satisfy the deadline constraint, which contradicts to the assumption. □

Accordingly, given a fixed j, we can check whether there exists a feasible $i(< j)$ to insert o_r as follows.

Corollary 1. *Given a fixed j, there exists a feasible i for inserting o_r if and only if (1) $picked[j] \leq K_w - K_r$, (2) $arr[j] + Dio[j] + dis(l_j, d_r) \leq e_r$, and (3) $Dio[j] + det(l_j, d_r, l_{j+1}) \leq slack[j]$.*

4.4.3 Algorithm Sketch

Algo. 3 illustrates the process of linear DP insertion. In line 4, we handle the cases when $i = j$ using the same way as in Algo. 2. In lines 5-7, we first check whether there exists a feasible i for the given j by Corollary 1. If yes, we calculate the minimal increased distance Δ_j^* and its corresponding i (*i.e.*, $Plc[j]$). In line 8, we prune according to Lemma 4 and dynamically update $Dio[j+1]$ and $Plc[j+1]$.

Example 2. *Back to the settings in Example 1. Suppose w_1 is assigned to serve r_1 following the route $S_{w_1} = \langle w_1, v_2, v_4 \rangle$. When r_2 is released at time 5, w_1 is at v_1. So there are another $n = 2$ vertices in S_{w_1} except for the current location of w_1. If we insert r_2 into the current route S_{w_1}, the arrays in line 2 of Algo. 3 are initialized as in Table 3. $ddl[0] = \infty$ since $l_0 = v_1$ is neither origin nor destination of r_1. $arr[0] = 5$ since current time is 5. $picked[0] = 0$ because w_1 picks up no requests. For $k > 0$, $ddl[k], arr[k], picked[k]$ are initialized using Eq. (6),Eq. (7) and Eq. (9). Specifically, $ddl[1] = e_{r_1} - dis(o_{r_1}, d_{r_1}) = 13, ddl[2] = e_{r_1} = 23$ since $l_1 = v_2$ is o_{r_1} and $l_2 = v_4$ is d_{r_1}. $arr[1] = arr[0] + dis(v_1, v_2) = 6, arr[2] = arr[1] + dis(v_2, v_4) = 16$. According to Eq. (8), $slack[\cdot]$ is calculated from $n = 2$ to 0. $slack[n]$ is always initialized with ∞ because all requests have been delivered at l_n. $slack[1] = \min\{slack[2], ddl[2] - arr[2]\} = 7, slack[0] = \min\{slack[1], ddl[1] - arr[1]\} = 7$. Then lines 3-9 of Algo. 3 work as follows. When $j = 0$, line 4 (*i.e.*, route $\langle v_1, o_{r_2}, d_{r_2}, v_2, v_4 \rangle$ as Fig. 2b) violates (4) of Lemma 4 because $\Delta_{i,j} = dis(v_1, o_{r_2}) + dis(o_{r_2}, d_{r_2}) + dis(d_{r_2}, v_2) - dis(v_1, v_2) = 25 > slack[1]$ according to Eq. (5) (case $i = j < n$). We then update $Dio[1] = \infty, Plc[1] = \infty$ since $det(v_1, o_{r_2}, v_2) = dis(v_1, o_{r_2}) + dis(o_{r_2}, v_2) - dis(v_1, v_2) = 15 > slack[0] = 7$ using Eq. (11) and Eq. (12). When $j = 1$, line 4 (*i.e.*, route $\langle v_1, v_2, o_{r_2}, d_{r_2}, v_4 \rangle$ as Fig. 2b) also violates (4) of Lemma 4 because $\Delta_{i,j} = dis(v_2, o_{r_2}) + dis(o_{r_2}, d_{r_2}) + dis(d_{r_2}, v_4) - dis(v_2, v_4) = 9 > slack[1]$. In line 5, Corollary 1 (2) is obviously violated because $Dio[1] = \infty$. We then update $Dio[2] = \min\{Dio[1], det(v_2, o_{r_2}, v_4)\} = 5$ because $picked[1] = 1 \leq K_w - K_r = 3$ and $det(v_2, o_{r_2}, v_4) = 5 \leq slack[1]$ in Eq. (11). $Plc[2] = 2 - 1 = 1$ since $Dio[1] > det(v_2, o_{r_2}, v_4)$ according to Eq. (12). Finally, when $j = 2$, line 4 ((*i.e.*, route $\langle v_1, v_2, v_4, o_{r_2}, d_{r_2} \rangle$ as Fig. 2a)) violates Lemma 4 (3) because $arr[2] + dis(v_4, o_{r_2}) + dis(o_{r_2}, d_{r_2}) = 33 > e_{r_2}$. In line 5, all conditions in Corollary 1 are satisfied. Thus, $\Delta_2^* = det(v_4, d_{r_2}, NIL) + Dio[2] = 8$ in line 6 and $\Delta^* = \Delta_2^* = 8, i^* = Plc[2] = 1, j^* = 2$ in line 7. In line 10, S^* becomes $\langle v_1, v_2, v_3, v_4, v_5 \rangle$ by inserting $o_{r_2} = v_3$ at $l_{i^*} = v_2$ and $d_{r_2} = v_5$ at $l_{j^*} = v_4$.*

Table 3: $arr, ddl, slack, picked, Dio, Plc$ in Example 2.

k	0	1	2
$ddl[k]$	∞	13	23
$arr[k]$	5	6	16
$picked[k]$	0	1	0

k	0	1	2
$slack[k]$	7	7	∞
$Dio[k]$	∞	∞	5
$Plc[k]$	NIL	NIL	1

Complexity Analysis. Line 2 and 10 take $O(n)$ time. There are $O(n)$ iterations in lines 3-9 and it only takes $O(1)$ time in each iteration. Thus, the total time cost is $O(n)$ and the memory cost is still $O(n)$. If a shortest distance query takes $O(q)$ time, the total time complexity is $O(nq)$.

5. INSERTION BASED SOLUTION

This section presents *pruneGreedyDP*, an efficient and effective solution to the URPSM problem leveraging the linear DP insertion. It consists of two phases. The first is a *decision phase* to decide whether to serve a new request or not. The second is a *planning phase* to add the request to be served into a route. We also propose a *pruning* strategy based on the results from the decision phase for route planning. Since shortest distance queries are important for applications on road networks, we also discuss how to minimize the usage of shortest distance queries.

5.1 Decision Phase

Since the unified cost defined in the URPSM problem consists of the total travel distance and the total penalty of unserved requests, it is reasonable to reject the request whose penalty is smaller than the increased distance if serving it. We propose a lower bound of the minimal increased distance as the metric to decide whether to serve a new request or not. The metric can be checked in $O(n)$ time and requires only one shortest distance query.

5.1.1 Calculating Lower Bound of Δ^*

We calculate the lower bound of Δ^* (denoted as LB_{Δ^*}) by adapting the calculation of $\Delta_{i,j}$ in linear DP insertion with Euclidean distance. The reasons can be reflected in three aspects.

- The Euclidean distance is usually smaller than the distance on a road network.
- It only takes $O(n)$ time to calculate Δ^* leveraging the techniques proposed in Sec. 4.4.
- We can use the auxiliary arrays *e.g.*, $arr[\cdot]$ for travel time to calculate shortest distances without any query.

LB_{Δ^*} is derived from the lower bound of detour (denoted as $ldet(,,)$) and the lower bound of $\Delta_{i,j}$ in Eq. (5) (denoted as $LB_{\Delta_{i,j}}$), as explained below.

Lower Bound of Detour. The detour when inserting l_j between l_i and l_k is bounded by:

$$det(l_i,l_j,l_k) \geq euc(l_i,l_j)+euc(l_j,l_k)-dis(l_i,l_k) = ldet(l_i,l_j,l_k)$$

Lower Bound of $\Delta_{i,j}$. Substituting $ldet(,,)$ into Eq. (5), we have the following lemma.

LEMMA 7. *Given a worker w and a new request r, $LB_{\Delta_{i,j}}$ can be calculated as*

$$LB_{\Delta_{i,j}} = \begin{cases} euc(l_n,o_r)+\mathcal{L}, & \text{if } i=j=n \\ euc(l_i,o_r)+\mathcal{L}+euc(d_r,l_{i+1})-(arr[i+1]-arr[i]), & \\ & \text{if } i=j<n \\ euc(l_i,o_r)+euc(o_r,l_{i+1})-(arr[i+1]-arr[i]) & \\ \quad +euc(l_j,d_r)+euc(d_r,l_{j+1}), & \\ \quad -(arr[j+1]-arr[j]) & \text{otherwise} \end{cases} \tag{15}$$

with only one shortest distance query, i.e., $\mathcal{L}=dis(o_r,d_r)$.

PROOF. When $i=j=n$, $LB_{\Delta_{i,j}}=euc(l_n,o_r)+\mathcal{L}$ without any extra shortest distance query.

When $i=j<n$, $LB_{\Delta_{i,j}}=euc(l_i,o_r)+\mathcal{L}+euc(d_r,l_{i+1})-dis(l_i,l_{i+1})$. Note that we cannot use any approximated distance which is smaller than the shortest distance. However, as the auxiliary array $arr[\cdot]$ indicates the arriving time of l_i, we have $dis(l_i,l_{i+1})=arr[i+1]-arr[i]$ if we expect travel time between l_i and l_{i+1}. Alternatively, an additional auxiliary array can be used to store the travel distance between two adjacent vertices in S_w. It then directly represents $dis(l_i,l_{i+1})$ if travel distance is used.

Similarly, we can also use $arr[i+1]-arr[i]$ and $arr[j+1]-arr[j]$ to substitute the shortest distances $dis(l_i,l_{i+1})$ and $dis(l_j,l_{j+1})$, respectively. Note that we only need one shortest distance query, *i.e.*, $\mathcal{L}=dis(o_r,d_r)$. ☐

Lower Bound of Δ^*. We calculate $LB_{\Delta^*}=\min\{LB_{\Delta_{i,j}}\}$ as follows. When $i=j$, we can calculate $\min\{LB_{\Delta_{i,j}}\}$ in linear time based on Eq. (15), since there are $O(n)$ possible cases. When $i<j$, we apply dynamic programming to calculate the minimal from all the possible $O(n^2)$ values.

Firstly, we use $arr[\cdot]$ to substitute the shortest distance.

$$ldet(l_j,d_r,l_{j+1})=euc(l_j,d_r)+euc(d_r,l_{j+1})-(arr[j+1]-arr[j])$$
$$ldet(l_{j-1},o_r,l_j)=euc(l_{j-1},o_r)+euc(o_r,l_j)-(arr[j]-arr[j-1])$$

Then we substitute both equations into Eq. (11). Dio_{euc} is the value of $Dio[\cdot]$ substituted with Euclidean distance.

$$Dio_{euc}[j] = \begin{cases} \infty, \text{ if } picked[j-1]>K_w-K_r \\ Dio_{euc}[j-1], \text{ if } ldet(l_{j-1},o_r,l_j)>slack[j-1] \\ \min\{Dio_{euc}[j-1], ldet(l_{j-1},o_r,l_j)\}, \text{ otherwise} \end{cases} \tag{16}$$

Finally, LB_{Δ^*} can be calculated as follows.

Algorithm 4: Decision Algorithm

input : α, workers W and a request r
output: a set of lower bound LB for each w

1 $\mathcal{L} \leftarrow dis(o_r,e_r), LB \leftarrow \emptyset$;
2 **foreach** $w \in W$ **do**
3 $\quad LB_{\Delta^*} \leftarrow$ `Linear DP Insertion(w, r)` by using Euclidean distance and $\mathcal{L}$ according to Lemma 7 and Algo. 3;
4 $\quad LB \leftarrow LB \cup \{(LB_{\Delta^*}, w)\}$;
5 **if** $p_r < \alpha \cdot \min LB$ **then** reject r;
6 **return** LB;

$$LB_{\Delta^*} = \min_{i\leq j, j=0,\cdots,n} \begin{cases} euc(l_n,o_r)+\mathcal{L}, & \text{if } i=j=n \\ euc(l_i,o_r)+\mathcal{L}+euc(d_r,l_{i+1}) & \\ \qquad -(arr[i+1]-arr[i]), & \text{if } i=j<n \\ det(l_j,d_r,l_{j+1})+Dio_{euc}[j] & \text{if } i<j \end{cases} \tag{17}$$

5.1.2 Algorithm Sketch

Algo. 4 illustrates the process of the decision process. For each worker in W, we calculate LB_{Δ^*} using Eq. (17). Note that we use LB to store all LB_{Δ^*} for each worker since we will use LB_{Δ^*} in the planning phase.

Time Complexity. Line 3 takes $O(|W|+|R|)$ time in total. Line 4 and 5 take $O(|W|)$ time. Thus Algo. 4 takes $O(|W|+|R|)$ time. If a shortest distance query takes $O(q)$ time, the total time complexity is $O(|W|+|R|+q)$.

5.2 Planning Phase

The planning phase first prunes candidate workers and greedily adds a new request into the route of the best worker.

5.2.1 Pruning Candidate Workers

Although many pruning strategies [13][25][18] have been proposed to filter candidate workers to serve the new request, they mostly rely on the duration of deadlines and grid indices. These strategies can 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 bounds (LB) from the decision phase. The pruning strategy is based on the following lemma.

LEMMA 8 (PRE ORDERED PRUNING). *Assume workers are already sorted according to LB_{Δ^*} in LB, which is calculated in the decision phase. If w_a is ahead of w_b in the sorted order and Δ^* of w_a is smaller than LB_{Δ^*} of w_b, we can safely ignore all workers after w_a.*

PROOF. Δ^* is the actual increased distance of worker w_a and LB_{Δ^*} is the lower bound of the actual increased distance of worker w_b. Since the workers are already sorted according to LB_{Δ^*}, Δ^* of w_a is also smaller than LB_{Δ^*} of any worker after w_b. ☐

Note that existing studies [30][25] often iterate all candidate workers with the actual shortest distances, making them time-consuming in practice. As we will show in Sec. 6, the pruning strategy based on Lemma 8 can save tens of billions of shortest distance queries when the deadline of request is long or when the number of workers is large.

5.2.2 Finding the Best Worker

After pruning, the next step is to find the worker with the minimal increased distance. In *pruneGreedyDP*, this is

Algorithm 5: pruneGreedyDP

input : α, workers W and requests R
output: a set of route $\{S_w\}$ and unified cost UC
1 Build grid index and initialize R^- with $\emptyset$;
2 **foreach** *new request* $r \in R$ **do**
    /* Phase 1: Decision */
3     $Cand \leftarrow$ filter the candidate according to grid index, deadline, etc.;
4     $LB \leftarrow$ `Decision`$(\alpha, Cand, r)$;
    /* Phase 2: Planning */
5   **if** r *is decided to be served* **then**
6         $w' \leftarrow NIL, \Delta^*_{w'} \leftarrow \infty$;
7   **foreach** (LB_{Δ^*}, w) *in sorted* LB **do**
8   **if** $\Delta^*_{w'} < LB_{\Delta^*}$ **then** break;
9             $\Delta^*_w \leftarrow$ `Linear DP Insertion`(w, r);
10   **if** $\Delta^*_w < \Delta^*_{w'}$ **then** $w' \leftarrow w, \Delta^*_{w'} \leftarrow \Delta^*_w$;
11   **if** w' *is not NIL* **then** update $S_{w'}$ and arr of w' accordingly;
12   **if** r *is rejected* **then** $R^- \leftarrow R^- \cup \{r\}$;
13 $UC \leftarrow \alpha \sum_{w \in W} D(S_w) + \sum_{r \in R^-} p_r$;
14 **return** $\{S_w\}$ and UC;

performed using the DP insertion. We use auxiliary array $arr[\cdot]$ to further reduce the times of shortest distance queries. Compared with existing work [34] which needs $3n$ shortest distance queries, we only need $2n+1$.

LEMMA 9. *Using the auxiliary array $arr[\cdot]$, the linear DP insertion only needs $2n+1$ shortest distance queries.*

PROOF. By replacing Euclidean distance with the shortest distance in Eq. (15), we can calculate Δ based on $arr[\cdot]$. Beyond the shortest distance query for $\mathcal{L} = dis(o_r, d_r)$, we only need the shortest distance between $o_r(d_r)$ and $l_1, \cdots, l_n$. Thus, it is $2n+1$ times in total. $\square$

5.3 Algorithm Sketch

Algo. 5 illustrates the *pruneGreedyDP* algorithm. In line 1, we build grid index and initialize R^-. For each new request, we first filter a set of candidate workers in line 3 and then start decision in line 4. If the request is decided to be served in line 4, we add it into a route in lines 5-11. Iterations in lines 7-10 are the implementation of our pruning strategy in Lemma 8. Specifically, we use linear DP insertion to calculate Δ^* for each w in line 9 and update the currently best worker w' with minimal increased distance ($\Delta^*_{w'}$) in line 10. If a feasible worker w' is found at the end of an iteration, we update $S_{w'}$ with 2 (in Fig. 2a), 3 (in Fig. 2b) or 4 (in Fig. 2c) shortest path queries together with the auxiliary array $arr[\cdot]$. Finally, we calculate the unified cost in line 13.

EXAMPLE 3. *Back to the settings in Example 1. Suppose that w_1 is assigned to serve r_1 at time 0 following route $S_{w_1} = \langle v_7(\text{location of } w_1), v_2, v_4\rangle$ and w_2 is assigned to serve r_2 at time 5 following route $S_{w_2} = \langle v_3(\text{location of } w_2), v_3, v_5, \rangle$. At time 11, both w_1 and w_2 travel to v_8 when r_3 is released. In line 3 of Algo. 5, $Cand = \{w_1, w_2\}$, which is the input of Algo. 4 to calculate a set of lower bounds LB. Note that in line 1 of Algo. 4, we query the shortest distance $\mathcal{L} = 4$ only once and use it to calculate LB_{Δ^*} with Euclidean distance of coordinates. According to Lemma 7, LB_{Δ^*} of w_1 is $euc(v_8, v_8) + \mathcal{L} + euc(v_5, v_4) - (arr_{w_1}[2] - arr_{w_1}[1]) = 0 + 4 + 2.8 - 5 = 1.8$, LB_{Δ^*} of w_2 is $euc(v_8, v_8) + \mathcal{L} + euc(v_5, v_5) - (arr_{w_2}[2] - arr_{w_2}[1]) = 4 - 4 = 0$. We have $LB = \{(1.8, w_1), (0, w_2)\}$. As the minimal lower bound is 0, which is less than the penalty $p_{r_3} = 9$, we decide to serve the request. Then in the planning phase, $(0, w_2)$ is first extracted in line 7 and $\Delta^*_{w_2} = 0$ in line 9. In line 8 of the second round, since $\Delta^*_{w'} = 0 < 1.8$, we just break the iteration. In line 13, $UC = D(S_1) + D(S_2) = 25$ when $\alpha = 1$.*

Table 4: Statistics of datasets.

Dataset	#(Requests)	#(Vertices)	#(Edges)
NYC	517,100	807,795	2,100,632
Chengdu	259,347	214,440	466,330

Time Complexity. There are $O(|R|)$ iterations in line 2. Line 3 takes $O(|W|)$ time and line 4 takes $O(|W|+|R|)$ time because there are $|W|$ workers and $|R|$ requests in total, and the DP insertion takes linear time (*i.e.*, $O(|R|)$ in total). Thus, the total time complexity of lines 3-4 is $O(|R|^2 + |R||W|)$. The sorting in line 7 takes $O(|W| \log |W|)$ time and lines 8-10 take $O(|W| + |R|)$ time for the same reason. Thus, the total time complexity of lines 5-11 is $O(|R|^2 + |R||W| \log |W|)$. Line 12 also takes $O(|W|+|R|)$ time. Thus, Algo. 5 takes $O(|R|^2 + |R||W| \log |W|)$ time. If a shortest path and distance query takes $O(q)$ time, the total time complexity is $O(|R|^2 q + |R||W|q + |R||W| \log |W|)$.

6. EXPERIMENTAL STUDY

This section presents the experimental evaluations of our proposed algorithms.

6.1 Experimental Setup

Datasets. We conduct experimental evaluations on two real citywide taxi datasets. The first is collected by Didi Chuxing [3] in Chengdu, China, which is published through its GAIA initiative [4]. The second is a public dataset [8] 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 [11][13][38][42]. We use the data from the day with the most requests for evaluation (November 18, 2016 in Chengdu, and April 09, 2016 in New York), which are denoted as *Chengdu* and *NYC*, respectively. Each tuple in the two datasets is a taxi request consisting of a pickup latitude/longitude, a drop off latitude/longitude and a release time. Since only *NYC* contains the request capacity K_r, we generate K_r for *Chengdu* according to its distribution in *NYC*. The road network of *NYC* is downloaded from Geofabrik [5]. For *Chengdu*, we use the latest city boundaries [1] and extract its road network out of the national road network of China from Geofabrik via Osmconvert [7]. Each road network is represented as an undirected graph. Table 4 summarizes the number of requests as well as the amounts of vertices and edges in each graph. *NYC* is the largest dataset among existing studies [25][17][42][18][30][11][13]. For instance, $|V|$ and $|E|$ in *NYC* are 6.6 and 11.1 times larger than those of the road network used in [25][17]. The number of requests in *NYC* is 47.7% more than that in [42]. *Chengdu* has a larger road network and comparable requests than existing literature.

Implementation. We simulate ride-sharing, a representative shared mobility application following the settings in [25][13]. The origin and the destination of each request are pre-mapped to the closest vertex in the road network. The initial location of a worker is randomly chosen from

Table 5: Parameter settings.

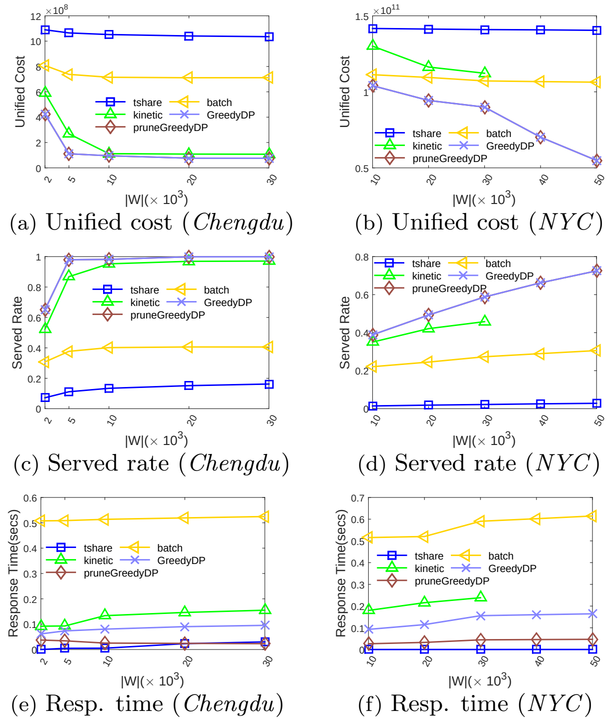
Parameters	Settings
Grid size g (kilometer)	**1**, 2, 3, 4, 5
Delivery deadline e_r (minute)	5, **10**, 15, 20, 25
Capacity K_w	3, **4**, 6, 10, 20
Weight α	**1**
Penalty p_r $(\times dis(o_r, d_r))$	*Chengdu*: 2, 5, **10**, 20, 30 *NYC*: 10, 20, **30**, 40, 50
Number of workers $\lvert W\rvert$	*Chengdu*: 2k, 5k, **10k**, 20k, 30k *NYC*: 10k, 20k, **30k**, 40k, 50k

the vertices in the road network. When a worker is serving a request, he/she follows the planned route and moves to the destination. Since a taxi usually travels with different speeds on different types of roads *e.g.*, 23 m/s in motorways or 6 m/s in residential streets, we assign a constant speed for each type of road *i.e.*, 80% of the maximum legal speed limit in their cities [2] and assume the taxi travels at the different speeds on different types of roads. Table 5 summarizes the major parameters of experiments. The default values are marked in bold. The delivery deadline is calculated as the release time of a request added by the parameter in the table. For example, the default deadline for a request with release time t_r is $t_r + 10min$. K_w is generated using a gaussian distribution with $\mu = 3, \cdots, 20$, because neither dataset specifies this information. We fix α to 1 so that the first term of $UC(W, R)$ in Eq. (1) is equivalent to the total travel distance. The penalty of a request is set by a parameter in the table multiplied by the shortest distance between origin and destination of the request, *e.g.*, $p_r = 10 \times dis(o_r, d_r)$ by default. Note that α is fixed to 1 and $p_r = 2, \cdots, 50(\times dis(o_r, d_r))$, which is equivalent to adjusting the proportion between c_r and c_w when maximizing the total revenue. Both p_r and $|W|$ of *NYC* are larger than *Chengdu* for its larger road network and number of drivers.

The experiments are conducted on a server with 40 Intel(R) Xeon(R) E5 2.30GHz processors with hyper-threading enabled and 128GB memory. The simulation implementation is single-threaded, and the total running time (the time to construct spatial index and labels for shortest path and distance query excluded) is limited to 10/20 hours for *Chengdu* and *NYC*. Based on the results of some work [25] on real-time ride-sharing, a real-time solution should stop before the time limitation. All the algorithms are implemented in GNU C++. Each experimental setting is repeated 30 times and the average results are reported. We only store the vertices and edges of the road network (*i.e.*, graph) through weighted adjacency list. The shortest distance and shortest path query are both on the fly, using a hub-based labeling algorithm implemented for road network [9]. An LRU cache [25] is maintained for shortest distance and path queries, and is used by all the algorithms.

Compared Algorithms. We compare *pruneGreedyDP* with the following state-of-the-art algorithms.

- ***tshare* [30].** It first filters workers via a searching process and then applies basic insertion to find a worker with minimal increased distance for each new request.
- ***kinetic* [25].** It uses a kinetic tree to maintain every possible route to serve all the remaining requests. Unlike *tshare*, the insertion operation is recursively executed based on the tree structure.
- ***batch* [11].** It first generates groups of requests in a batch (*e.g.*, 6 seconds) and sorts the groups. Then it greedily assigns requests in each group by inserting each request into the route of current workers, and finally chooses the worker who can serve more requests with minimal increased distance.
- ***GreedyDP*.** It is a variant of our *pruneGreedyDP* algorithm without the pruning strategy in Lemma 8.



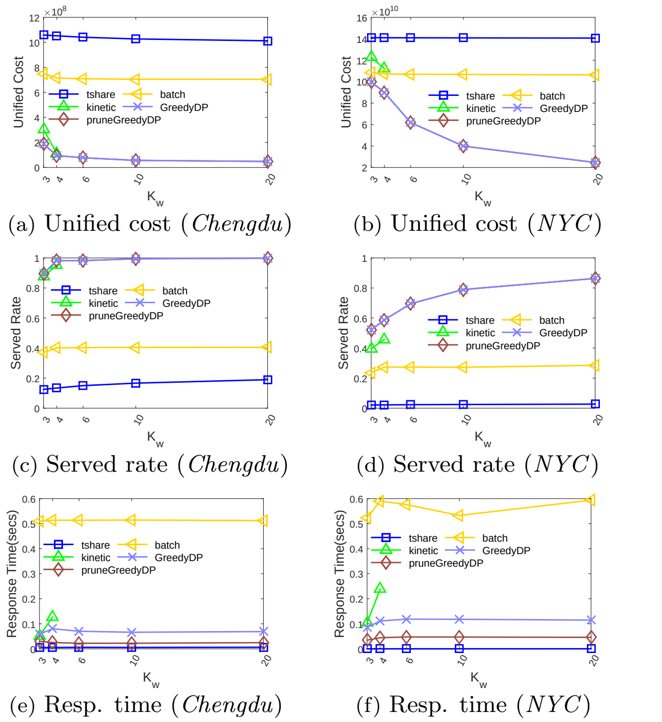
(a) Unified cost (*Chengdu*) (b) Unified cost (*NYC*)
(c) Served rate (*Chengdu*) (d) Served rate (*NYC*)
(e) Resp. time (*Chengdu*) (f) Resp. time (*NYC*)

Figure 3: Performance of varying number of workers $|W|$.

Metrics. All the algorithms are evaluated in terms of total unified cost, served rate ($|R^+|/|R|$) and response time (average waiting time to process a single request, resp. time for short). Served rate and response time are the metrics in many large-scale real-time ride-sharing proposals [30][25]. We also assess the memory cost of each algorithm. Note that the memory usage of auxiliary arrays can be omitted compared to the size of the graph, cache and grid index. Since the memory cost of graph and cache is constant for every algorithm, we only evaluate the memory cost of grid index when varying the size of grid g. We also evaluate the number of saved shortest distance query (distance query for short) between *pruneGreedyDP* and *GreedyDP* to show the effectiveness of the pruning strategy.

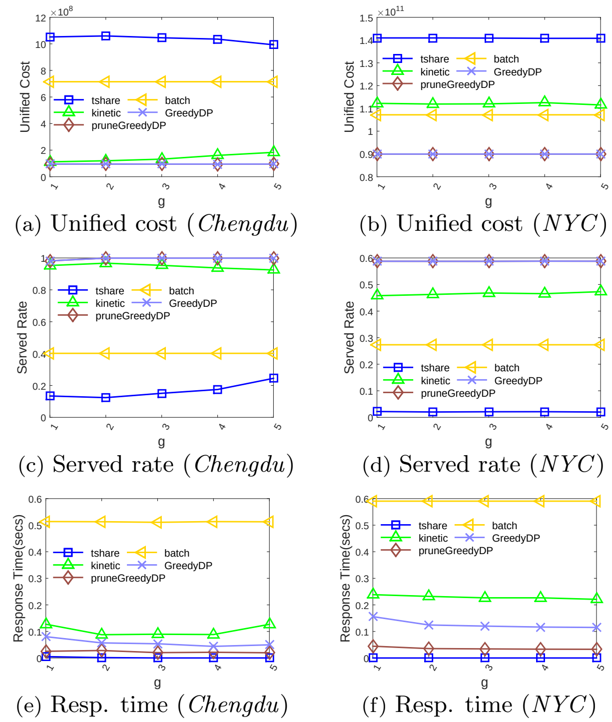
6.2 Experimental Results

Impact of Number of Workers $|W|$. Fig. 3 presents the results of varying the number of workers. Overall, *pruneGreedyDP* outperforms the rest in terms of unified cost by 12.41% to 85.36% on *Chengdu* and *NYC*. The unified costs of all the algorithms decrease with the increasing number of workers, because more requests can be served. For the same reason, the served rates of all the algorithms increase on both datasets. *pruneGreedyDP* has the highest served rate, 54.94% and 141.61% higher than *batch*. The results of served rate in *Chengdu* indicate that *pruneGreedyDP* is competitive with *kinetic* and better than *batch* when maximizing the number of served requests. For a larger road network, the



(a) Unified cost (*Chengdu*) (b) Unified cost (*NYC*)
(c) Served rate (*Chengdu*) (d) Served rate (*NYC*)
(e) Resp. time (*Chengdu*) (f) Resp. time (*NYC*)

Figure 4: Performance of varying capacity K_w.



(a) Unified cost (*Chengdu*) (b) Unified cost (*NYC*)
(c) Served rate (*Chengdu*) (d) Served rate (*NYC*)
(e) Resp. time (*Chengdu*) (f) Resp. time (*NYC*)

Figure 5: Performance of varying grid size g.

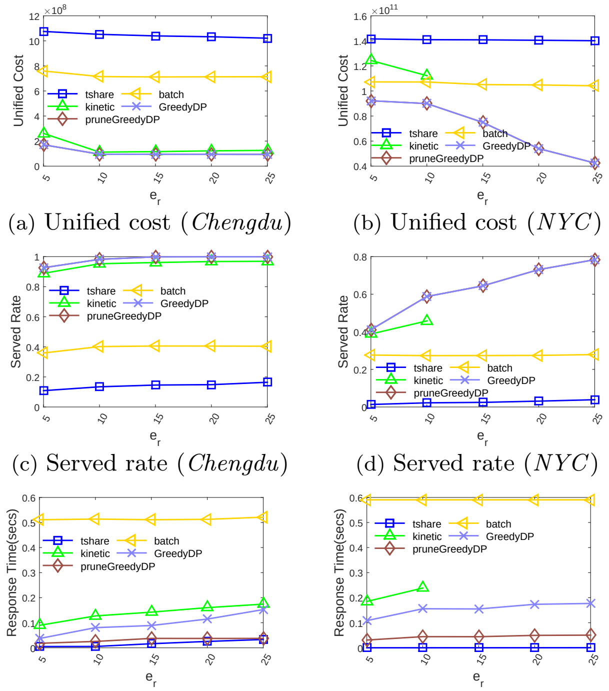
served rates of all algorithms dramatically decrease, which aligns with Lemma 1, *i.e.*, the number of served requests is affected by $|V|$. The response time of all baselines increase with the increase of $|W|$. *tshare* is the fastest because its searching process mistakenly removes many possible workers, which leads to the lowest served rate (from 1%to 16%). *pruneGreedyDP* is the second fastest, 2.46 to 32.08 times faster than *kinetic* and *batch*. Note that *kinetic* fails to finish the simulation in 20 hours when $|W|$ is 40k and 50k. By using the proposed pruning strategy, the total number of saved shortest distance query increases from 5.27 billion to 42.20 billion in *NYC*, and from 22.26 billion to 45.16 billion in *Chengdu*. Thus, the response time of *pruneGreedyDP* is 2.76 times faster than *GreedyDP* in average, validating the efficiency of our pruning strategy.

Impact of Capacity of Workers K_w. Fig. 4 shows the results of varying the capacity of workers. With a larger capacity, all the algorithms incur a lower unified cost on *Chengdu*. Our *pruneGreedyDP* algorithm has a unified cost up to 71% lower than the others. *kinetic* fails to stop in case of a large K_w because of its exponential time complexity $(2K_w)!$ [17]. In contrast, *batch* is more stable with a slight decrease in unified cost. In terms of served rate, *pruneGreedyDP* is still the best, outperforming the others by up to 96%. In terms of response time, *tshare* is the fastest for the same reason as varying the number of workers. The reduction in response time of *pruneGreedyDP* over *kinetic* and *batch* is 41% to 95% times on *Chengdu* and 47% to 93% times on *NYC*.

Impact of Grid Size g. Fig. 5 plots the results of varying the grid size g. In terms of unified cost, both *kinetic* and *tshare* are relatively insensitive to the change of grid size, and *pruneGreedyDP* outputs the lowest unified cost on both datasets. In terms of served rate, *batch* almost constantly yields 40.1% on *Chengdu* and 25.7% on *NYC*. *pruneGreedyDP* achieves the highest served rate on both datasets, 3.0% and 87.6% higher than the baselines on *Chengdu*, 16.9% and 96.5% higher on *NYC*. Again, *tshare* has the shortest response time yet extremely low served rate. *kinetic* and *batch* are up to 3.11 and 25.98 times slower than *pruneGreedyDP* in terms of response time. For the memory usage of grid index, *tshare* consumes the most from 609.46 to 5.38 MB in *NYC* and 9389.72 to 326.98 MB in *Chengdu* while others consume up to 0.30 MB for *Chengdu* and 3.23 MB for *NYC*. This is because the grid index of the other algorithms only stores the IDs of workers in the grid instead of a sorted list of grids like *tshare*.

Impact of Deadline e_r. Fig. 6 shows the results of varying the deadline e_r. With a larger deadline, the unified costs of all the algorithms decrease while the served rates of all the algorithms increase. The reason is that a longer deadline allows more requests to be served, and thus a lower unified cost and a higher served rate. In terms of effectiveness (unified cost and served rate), *pruneGreedyDP* is still the best. Note that when $\alpha = 1$ and the served rate closed to 100%, the unified cost approximates the total travel distance. Thus *pruneGreedyDP* yields a smaller travel distance than *kinetic* [25] and *tshare* [30]. The response time of *batch* and *pruneGreedyDP* is stable while that of the others notably increases. The increase of *GreedyDP* is 1.63 and 4.12 times in both datasets when e_r increases from 5 to 25 minutes. Conversely, the response time of *pruneGreedyDP* remains within 50ms. This is because when varying e_r, 24.95 to 83.99 billions of shortest distance queries are saved in *Chengdu* and 16.43 to 57.90 billions are saved in *NYC* using the new pruning strategy.

Impact of Penalty p_r. Fig. 7 presents the results of varying the penalty. The unified costs of all the baselines increase with the penalty while that of *pruneGreedyDP* is always the smallest. This indicates that *pruneGreedyDP* actu-

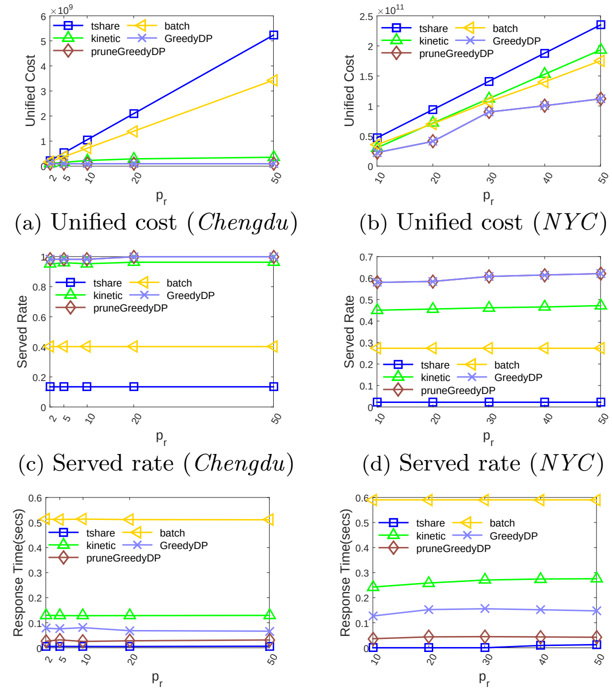


(a) Unified cost (*Chengdu*) (b) Unified cost (*NYC*)

(c) Served rate (*Chengdu*) (d) Served rate (*NYC*)

(e) Resp. time (*Chengdu*) (f) Resp. time (*NYC*)

Figure 6: Performance of varying deadline e_r.



(a) Unified cost (*Chengdu*) (b) Unified cost (*NYC*)

(c) Served rate (*Chengdu*) (d) Served rate (*NYC*)

(e) Resp. time (*Chengdu*) (f) Resp. time (*NYC*)

Figure 7: Performance of varying penalty p_r.

ally remains competitive when maximizing the total revenue when the proportion between c_r and c_w varies. The served rate of *kinetic* slightly increases because a higher penalty may force it to try to serve more requests *i.e.*, p_r is larger than $\alpha \cdot \min LB$ in the decision phase. In terms of response time, our proposed algorithm is 4.17 and 16.34 times faster than *kinetic* and *batch*.

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

- Our *pruneGreedyDP* algorithm usually achieves a unified cost 1.2 to 12.8 times lower than the three state-of-the-art algorithms [25][11], while being able to serve more requests (at least 9% higher) in large-scale datasets. These results validate the effectiveness of our solution in route planning with multiple objectives.
- The algorithms with DP-based insertion, *GreedyDP* and *pruneGreedyDP*, are 2.62 to 20.72 times faster than the state-of-the-arts [25][11]. In many cases, *pruneGreedyDP* is 2.8 times faster than *GreedyDP*, due to tens of billions of shortest distance queries saved.
- Among the state-of-the-arts, *kinetic* [25] often fails to halt in time on large-scale datasets for its exponential time complexity. *batch* [11] is less effective and efficient than our solution but more scalable than *kinetic* in large-scale datasets. *tshare* [30] always has the fastest response time, but has the lowest served rate and the highest unified cost.

7. 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. We prove that there is no polynomial-time algorithm with constant competitive ratio to solve the URPSM problem and its variants proposed in previous studies. Since insertion is a basic yet ineffective operation in many existing solutions to route planning, we develop a novel dynamic programming based algorithm, which reduces the time complexity of insertion from cubic or quadric time to linear time. We then devise an effective and efficient two-phased solution leveraging the above DP-based insertion algorithm to address the URPSM problem approximately. Extensive experiments on real datasets show that our proposed solution outperforms the state-of-the-arts 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.

Acknowledgment

We are grateful to anonymous reviewers for their constructive comments. Yongxin Tong and Ke Xu's works are partially supported by the National Science Foundation of China (NSFC) under Grant No. 61502021 and 71531001, National Grand Fundamental Research 973 Program of China under Grant 2014CB340300, the Science and Technology Major Project of Beijing under Grant No. Z171100005117001 and Didi Gaia Collborative Research Funds for Young Scholars. Yuxiang Zeng and Lei Chen's works are partially supported by the Hong Kong RGC GRF Project 16207617, the National Grand Fundamental Research 1090 973 Program of China under Grant 2014CB340303, the National Science Foundation of China (NSFC) under Grant No. 61729201, the Science and Technology Planning Project of Guangdong Province, China, No. 2015B010110006, Microsoft Research Asia Collaborative Research and HKUST SSTP under Project FP305.

8. REFERENCES

[1] Boundary of Chengdu. https://www.openstreetmap.org/node/244077729.

[2] Default Speed Limits in OpenStreetMap. https://wiki.openstreetmap.org/wiki/OSM_tags_for_routing/Maxspeed.

[3] Didi Chuxing. http://www.didichuxing.com/.

[4] GAIA. https://outreach.didichuxing.com/research/opendata/.

[5] Geofabrik. https://download.geofabrik.de/.

[6] OpenStreetMap. http://www.openstreetmap.com/.

[7] Osmconvert. https://wiki.openstreetmap.org/wiki/Osmconvert.

[8] TLC Trip Record Data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[9] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, pages 230–241, 2011.

[10] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang. Optimization for dynamic ride-sharing: A review. *European Journal of Operational Research*, 223(2):295–303, 2012.

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

[12] N. Ascheuer, S. O. Krumke, and J. Rambau. Online dial-a-ride problems: Minimizing the completion time. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 639–650, 2000.

[13] M. Asghari, D. Deng, C. Shahabi, U. Demiryurek, and Y. Li. Price-aware real-time ride-sharing at scale: an auction-based approach. In *SIGSPATIAL*, pages 3:1–3:10, 2016.

[14] M. Asghari and C. Shahabi. An on-line truthful and individually rational pricing mechanism for ride-sharing. In *SIGSPATIAL*, pages 7:1–7:10, 2017.

[15] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 2005.

[16] M. Charikar and B. Raghavachari. The finite capacity dial-a-ride problem. In *FOCS*, pages 458–467, 1998.

[17] L. Chen, Q. Zhong, X. Xiao, Y. Gao, P. Jin, and C. S. Jensen. Price-and-time-aware dynamic ridesharing. In *ICDE*, pages 1061–1072, 2018.

[18] P. Cheng, H. Xin, and L. Chen. Utility-aware ridesharing on road networks. In *SIGMOD*, pages 1197–1210, 2017.

[19] B. Cici, A. Markopoulou, and N. Laoutaris. Designing an on-line ride-sharing system. In *SIGSPATIAL*, pages 60:1–60:4, 2015.

[20] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[21] P. M. d'Orey, R. Fernandes, and M. Ferreira. Empirical evaluation of a dynamic and distributed taxi-sharing system. In *ITSC*, pages 140–146, 2012.

[22] E. Feuerstein and L. Stougie. On-line single-server dial-a-ride problems. *Theoretical Computer Science*, 268(1):91–105, 2001.

[23] A. Gupta, M. Hajiaghayi, V. Nagarajan, and R. Ravi. Dial a ride from k-forest. *ACM Transactions on Algorithms (TALG)*, 6(2):41, 2010.

[24] W. M. Herbawi and M. Weber. A genetic and insertion heuristic algorithm for solving the dynamic ridematching problem with time windows. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 385–392, 2012.

[25] Y. Huang, F. Bastani, R. Jin, and X. S. Wang. Large scale real-time ridesharing with service guarantee on road networks. *PVLDB*, 7(14):2017–2028, 2014.

[26] H. Hung, R. Chapman, W. Hall, and E. Neigut. A heuristic algorithm for routing and scheduling dial-a-ride vehicles. In *ORSA/TIMS National Meeting*, 1982.

[27] J. J. Jaw. *Solving large-scale dial-a-ride vehicle routing and scheduling problems*. PhD thesis, Massachusetts Institute of Technology, 1984.

[28] J. J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. M. Wilson. A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. *Transportation Research Part B Methodological*, 20(3):243–257, 1986.

[29] A. Kleiner, B. Nebel, and V. A. Ziparo. A mechanism for dynamic ride sharing based on parallel auctions. In *IJCAI*, volume 11, pages 266–272, 2011.

[30] S. Ma, Y. Zheng, and O. Wolfson. T-share: A large-scale dynamic taxi ridesharing service. In *ICDE*, pages 410–421, 2013.

[31] S. Ma, Y. Zheng, and O. Wolfson. Real-time city-scale taxi ridesharing. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1782–1795, 2015.

[32] R. Mole and S. Jameson. A sequential route-building algorithm employing a generalised savings criterion. *Journal of the Operational Research Society*, 27(2):503–511, 1976.

[33] M. Ota, H. Vo, C. Silva, and J. Freire. A scalable approach for data-driven taxi ride-sharing simulation. In *Big Data*, pages 888–897, 2015.

[34] M. Ota, H. Vo, C. Silva, and J. Freire. Stars: Simulating taxi ride sharing at scale. *IEEE Transactions on Big Data*, PP(99):1–1, 2017.

[35] D. Pelzer, J. Xiao, D. Zehe, M. H. Lees, A. C. Knoll, and H. Aydt. A partition-based match making algorithm for dynamic ridesharing. *IEEE Transactions on Intelligent Transportation Systems*, 16(5):2587–2598, 2015.

[36] S. Roy and U. de Montréal. Centre de recherche sur les transports. *The construction of routes and schedules for the transportation of the handicapped*. Montréal: Université de Montréal, Centre de recherche sur les transports, 1983.

[37] Z. B. Rubinstein, S. F. Smith, and L. Barbulescu. Incremental management of oversubscribed vehicle schedules in dynamic dial-a-ride problems. In *AAAI*, pages 1809–1815, 2012.

[38] P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti. Quantifying the benefits of vehicle pooling with shareability networks. *Proceedings of the National Academy of Sciences*, 111(37):13290–13294, 2014.

[39] D. O. Santos and E. C. Xavier. Dynamic taxi and ridesharing: A framework and heuristics for the optimization problem. In *IJCAI*, volume 13, pages 2885–2891, 2013.

[40] S. Shaheen, A. Cohen, and I. Zohdy. Shared mobility: Current practices and guiding principles. Technical report, 2016.

[41] SUMC. What is shared-use mobility? `https://goo.gl/3Jw6z7`, 2018.

[42] R. S. Thangaraj, K. Mukherjee, G. Raravi, A. Metrewar, N. Annamaneni, and K. Chattopadhyay. Xhare-a-ride: A search optimized dynamic ride sharing system with approximation guarantee. In *ICDE*, pages 1117–1128, 2017.

[43] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu. A unified approach to route planning for shared mobility. `http://home.cse.ust.hk/~yzengal/yzengal_full.pdf`, 2018.

[44] N. H. Wilson, R. Weissberg, B. Higonnet, and J. Hauser. Advanced dial-a-ride algorithms. Technical report, 1975.

[45] N. H. Wilson, R. W. Weissberg, and J. Hauser. Advanced dial-a-ride algorithms research project. Technical report, 1976.

[46] A. C. C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *FOCS*, pages 222–227, 1977.

[47] S. Yeung, E. Miller, and S. Madria. A flexible real-time ridesharing system considering current road conditions. In *MDM*, volume 1, pages 186–191, 2016.

[48] Y. Zheng. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):29, 2015.