A Unified Approach to Route Planning for Shared Mobility

Yongxin Tong † Yuxiang Zeng ‡ Zimu Zhou †† Lei Chen ‡‡ Jieping Ye †† Ke Xu ††
† SKLSDE Lab, BBDC, 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,ke xu}@buaa.edu.cn, zzhou@tik.ee.ethz.ch, yzengal, leichen}@cse.ust.hk, 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 practicality 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 efficient and effective 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:
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 enabling 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
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 calculate 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 cubic 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 \(c(u, v)\).

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 \(W = \{w_1, \ldots, w_{|W|}\}\) to denote all the workers.
 Definition 3 (Request). A request is denoted by \( r = (o, d, t, e, p) \) with an origin \( o \in V \), a destination \( d \in V \), and a capacity \( K_r \). It is released on the shared mobility platform (platform for short) at a release time \( t \), and needs to be served before a deadline \( e \). A request is served if (i) a worker picks up \( r \) at \( o \) after \( t \); and (ii) the same worker drops \( r \) at \( d \), before \( e \). 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 \( e \) usually suffices since the deadline for pickup can be expressed as \( e_r = \text{dis}(o, d) \). 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 \( R = \{r_1, \ldots, r_n\} \) 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 \( S_w = \{o_w, l_w^1, \ldots, l_w^{|S_w|-1}, d_w\} \), where \( \{l_w^1, \ldots, l_w^{|S_w|-1}\} \) is an ordered sequence of origin and destination of \( R_w \), i.e., \( l_w^1 \in \{o | r \in R_w\} \cup \{d | r \in R_w\} \). A route is feasible if (i) \( v \in R_w, o, d, \) and \( o \) precedes \( d \) in the sequence; (ii) \( v \in R_w \), the time when \( w \) arrives at \( d \) is no later than the deadline \( e_d \); (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.,
\[
D(S_w) = \text{dis}(o_w, l_w^1) + \sum_{i=2}^{|S_w|-1} \text{dis}(l_{w}^{i-1}, l_{w}^{i})
\]

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 release time, and a weight coefficient \( \alpha \), 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
\[
UC(W, R) = \alpha \sum_{w \in W} D(S_w) + \sum_{r \in R^-} p_r
\]

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 \( r_1, r_2, r_3 \). The initial locations of workers are labeled on a road network with eight vertices \( v_1, \ldots, v_8 \) 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 \((t_r)\), 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 \((o_1, v_2, v_4)\), which reaches \( v_4 \) at time \( 5 + 1 + 5 + 5 = 16 \). Specifically, \( w_1 \) starts from \( v_2 \) and first travels from \( v_1 \) to \( v_2 \). \( w_1 \) picks up \( r_1 \) at \( v_2 \) and then travels from \( v_2 \) 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.

Next we show that many previous studies are special cases of our URPSM problem with specific \( \alpha \) 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 related to the travel distance and the fare for unit distance \( c_r \). Then the total revenue of the platform is calculated as:
\[
\text{revenue}(W, R) = c_r \sum_{r \in R^-} \text{dis}(o_r, d_r) = c_w \sum_{w \in W} D(S_w)
\]

(2)

Set \( \alpha = c_w \) and \( \forall r \in R, p_r = c_r \times \text{dis}(o_r, d_r) \):
\[
UC(W, R) = c_w \sum_{w \in W} D(S_w) + c_r \sum_{r \in R^-} \text{dis}(o_r, d_r)
\]

(3)

Substitute \( R^+ = R - R^- \) and Eq. (3) into Eq. (2):
\[
\text{revenue}(W, R) = c_r \sum_{r \in R} \text{dis}(o_r, d_r) - UC(W, R)
\]

(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.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R,W$</td>
<td>a set of requesters and workers</td>
</tr>
<tr>
<td>$o_r,d_r$</td>
<td>origin and destination of request $r$</td>
</tr>
<tr>
<td>$p_r$</td>
<td>penalty of each unserved request $r$</td>
</tr>
<tr>
<td>$R^l,R^u$</td>
<td>a set of served requests and rejected requests</td>
</tr>
<tr>
<td>$S(w),D(S(w))$</td>
<td>schedule of worker $w$ and its distance</td>
</tr>
<tr>
<td>$K_w,K_c$</td>
<td>capacity of worker $w$, capacity of request $r$</td>
</tr>
<tr>
<td>$s(.,.)$</td>
<td>shortest distance between two vertices</td>
</tr>
</tbody>
</table>

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 have 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 $\forall r \in R, p_r = c_r \times \text{dis}(o_r,d_r)$;
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., $\infty$). Then applying Yao’s Principle [46], no randomized algorithm has a constant competitive ratio.

The distribution $\chi$ 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 $d_r = o_r$, $c_r = t_r + \epsilon$, $\epsilon > 0$ 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., $E_r[OPT] = 0$.

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 $\epsilon$, $ALG$ is able to serve $r$ with a probability $\leq \frac{\epsilon}{|V|}$. Since there is only one request, the expected number of unserved requests of $ALG$ is $E_r[ALG] \geq 1 - \frac{\epsilon}{|V|}$. Hence

$$E_r[ALG] = \frac{1}{|V|} \rightarrow \infty$$

The above ratio becomes unbounded.

**Lemma 2.** When $\alpha = c_w$ and $\forall r \in R, p_r = c_r \times \text{dis}(o_r,d_r)$, 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 $\text{dis}(o_r,d_r) = |V|/2$, 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

$$E_r[OPT] \leq c_w |V|$$

$$E_r[ALG] \geq (1 - \frac{2\epsilon}{|V|}) \cdot p_r = (1 - \frac{2\epsilon}{|V|}) \cdot c_r \cdot |V|$$

If $\epsilon$ is small enough, then

$$E_r[ALG] \geq \frac{c_r}{2c_w} \left(1 - \frac{2\epsilon}{|V|}\right) = \Omega(\frac{c_r}{c_w})$$

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

$$E_r[OPT] \leq \alpha|V| = |V|$$

$$E_r[ALG] \geq \left(1 - \frac{2\epsilon}{|V|}\right) \cdot p_r$$

$$E_r[ALG] \geq \frac{p_r}{|V|} \left(1 - \frac{2\epsilon}{|V|}\right)$$

By setting a sufficiently small $\epsilon$ and $p_r = \infty$, the above ratio becomes unbounded, i.e., $E_r[ALG] / E_r[OPT] \rightarrow \infty$.

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
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 quadratic time complexity and linear memory complexity, and an improved version of 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 \( \Delta^* \). We update \( S^* \) using \( S'_w \) and \( \Delta^* \) 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 \( \Delta^* \), 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 \( O(n^3q) \) time.

### 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 \( l_i \) between \( l_i \) and \( l_k \). Specifically, the detour \( det(l_i, l_j, l_k) \) is defined as follows.

\[
\begin{align*}
\text{det}(l_i, l_j, l_k) &= \text{dis}(l_i, l_j) + \text{dis}(l_j, l_k) - \text{dis}(l_i, l_k) \\
&= \text{dis}(l_i, o_r) + \text{dis}(o_r, d_r) + \text{dis}(d_r, l_i) - \text{dis}(l_i, l_i), \\
&= \text{dis}(l_i, o_r, l_i) + \text{det}(l_i, d_r, l_i), \\
&= \text{dis}(l_i, o_r, l_i) + \text{det}(d_r, l_i, l_i), \\
&= \text{dis}(l_i, o_r, l_i) + \text{det}(l_i, l_i, l_i)
\end{align*}
\]

(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 \( ddl[k] = e_r - \text{dis}(o_r, d_r) \) if \( l_k \) is the origin of \( r \) and \( e_r \) if \( l_k \) is the destination.

\[
\begin{align*}
\text{ddl}[k] &= \begin{cases} 
\text{dis}(o_r, d_r), & \text{if } l_k \text{ is } o_r \\
\text{e}_r, & \text{if } l_k \text{ is } d_r
\end{cases}
\end{align*}
\]

(6)

Denote \( arr[k] \) as the time when \( w \) arrives at \( l_k \). Hence, \( arr[k] = arr[k - 1] + \text{dis}(l_{k-1}, l_k) \)

(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 $\text{ddl, arr, slack}$, 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 $(2)$ of Lemma 5 is violated then continue;
6    foreach $j \leftarrow i$ to $n$ do
7       if $j > i$ and $(3)$ of Lemma 4 is violated then break;
8       if $(4)$ of Lemma 4 is violated then continue;
9       if $(5)$ of Lemma 5 is violated then continue;
10      if $\Delta^* \leftarrow \Delta^*_{i,j}$, $i^* \leftarrow i$, $j^* \leftarrow j$;
11     if $\Delta^* < \infty$ then
12        $S' \leftarrow$ insert $x_i$ at $i^*$-th and $x_j$ 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 $l_k, \ldots, l_n$. slack[$k$] should be no larger than the deadline of location $l_{k+1}$ (min $\text{arr}[k] + 1$), and all the deadlines after location $l_{k+1}$ (slack[$k$] + 1). Hence, $\text{slack}$ can be calculated as follows.

$$\text{slack}[k] = \min_{k \geq k} (\text{arr}[k] - \text{arr}[k])$$

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

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

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

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

**Proof.** Since $o_r$ is inserted at $i$-th, $\text{picked}[k]$ should be no greater than $K_w - K_r$. As the request $r$ is delivered right after $l_k$, this will exceed the worker capacity $K_w$ at $l_k$ if $\text{picked}[k] > K_w - K_r$, which violates the capacity constraint of $w$.

**4.3.3 Algorithm Sketch**

**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 $\Delta^*$ (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 $\Delta^*_{j}$ as the minimal increased distance for a given $j$. For the general case in Fig. 2c, $\Delta_{j} = \min_{k > i} \Delta_{i,k}$.

$$\Delta_{j} = \min_{i,j} \left( \text{det}(l, o_r, l_{i+1}) + \text{det}(l_j, d_r, l_{j+1}) \right)$$

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, $\text{Dio}[j] = \min_{i < j} \text{det}(l_i, o_r, l_{i+1})$. $\text{Dio}[j]$ can be calculated via the following DP formulation.

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

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].

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

If Dio[j] and Plc[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 \( j = 0 \) to \( n \) do
   4. Update \( \Delta^*, i^*, j^* \) with special cases as shown in Fig. 2a and Fig. 2b;
   5. if \( j > 0 \) and Corollary 1 is satisfiable then
      6. \( \Delta^*_j = det(l_j, d_j, l_{j+1}) + Dio[j]; \)
      7. if \( \Delta^*_j < \Delta^* \) then
         \( \Delta^* \leftarrow \Delta^*_j, i^* \leftarrow Dio[j], j^* \leftarrow j; \)
      8. if \( arr[j] + dis(o_i, e_r) > e_r \) then break;
    9. Update Dio\([j+1]\) and Plc\([j+1]\) according to Eq. (11) and Eq. (12);
10. if \( \Delta^* < \infty \) then
    11. \( S' \leftarrow \text{insert } o_i \text{ at } i^*\text{-th and } d_r \text{ at } j^*\text{-th in } S_w; \)
12. 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

\[
\begin{align*}
det(l_{i'}, o_{j'}, l_{i'+1}) + det(l_j, d_j, l_{j+1}) &\leq slack[j] \quad (13) \\
det(l_{i'}, o_{j'}, l_{i'+1}) + det(l_j, d_j, l_{j+1}) &\leq e_r - arr[j] \quad (14)
\end{align*}
\]

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

Accordingly, given a fixed \( j \), we can check whether there exists a feasible \( i(\leq j) \) to insert \( o_i \) as follows.

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

4.4.3 Algorithm Sketch

Alg. 3 illustrates the process of linear DP insertion. In line 4, we handle the cases when \( i = j \) using the same way as in Alg. 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 \( \Delta^*_j \) 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 \( S_{w_1} = (w_1, v_2, r_1) \). When \( r_2 \) is released at time \( 5 \), \( w_2 \) is at \( v_3 \). 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 Alg. 3 are initialized as shown 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, \( ddl[1] = e_{v_1} - dis(o_1,d_{v_1}) = 13, ddl[2] = e_{v_1} - \max\{arr[0] + dis(v_1,v_2) = 6, arr[2] = 2\} = 16 \) according to Eq. (8). We then calculate \( slack[k] \) and \( Dio[k] \) as follows.

<table>
<thead>
<tr>
<th>( k )</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ddl[k] )</td>
<td>( \infty )</td>
<td>13</td>
<td>23</td>
</tr>
<tr>
<td>( arr[k] )</td>
<td>5</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>( picked[k] )</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

5. INSERTION BASED SOLUTION

This section presents \textit{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 un-served 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 $\Delta^*$

We calculate the lower bound of $\Delta^*$ (denoted as $LB_{\Delta^*}$) 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 $\Delta^*$ leveraging the techniques proposed in Sec. 4.4.
- We can use the auxiliary arrays $e.g.,~arr[i]$ for travel time to calculate shortest distances without any query.

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

Lower Bound of $\Delta_{i,j}$. Substituting $lbdet(\cdot,\cdot)$ 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

$$
LB_{\Delta_{i,j}} = \begin{cases} 
  euc(l_{i}, o_{i}) + L, & \text{if } i = j = n \\
  euc(l_{i}, o_{i}) + L + euc(d_{i}, l_{i+1}) - (arr[i+1] - arr[i]), & \text{if } j < n \\
  euc(l_{i}, o_{i}) + euc(o_{i}, l_{i+1}) - (arr[i+1] - arr[i]) + euc(l_{j}, d_{j}) + euc(d_{j}, l_{j+1}), & \text{if } j = n \\
  -(arr[j] - arr[j]) & \text{otherwise}
\end{cases}
$$

with only one shortest distance query, i.e., $L = dis(o_{i}, d_{i})$.

**Proof.** When $i = j = n$, $LB_{\Delta_{i,j}} = euc(l_{i}, o_{i}) + L$ without any extra shortest distance query.

When $i = j < n$, $LB_{\Delta_{i,j}} = euc(l_{i}, o_{i}) + L + euc(d_{i}, l_{i+1}) - dis(l_{i}, l_{i+1})$. Note that we cannot use any approximated distance which is smaller than the shortest distance. However, as the auxiliary array $arr[i]$ 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 $S_w$. Then it 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., $L = dis(o_{i}, d_{i})$.

Lower Bound of $\Delta^*$. 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 $arr[i]$ to substitute the shortest distance.

$$
lbdet(l_{i}, d_{i}, l_{j+1}) = euc(l_{i}, d_{j}) + euc(d_{j}, l_{j+1}) - (arr[j+1] - arr[j])
lbdet(l_{i-1}, o_{i}, l_{j}) = euc(l_{i-1}, o_{i}) + euc(o_{i}, l_{j}) - (arr[j] - arr[j-1])
$$

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

$$
Dio_{w,c}[j] = \begin{cases} 
  \infty, & \text{if picked}[j-1] > K_w - K_r \\
  Dio_{w,c}[j-1], & \text{if } lbdet(l_{j-1}, o_{i}, l_{j}) > slack[j-1] \\
  \min\{Dio_{w,c}[j-1], lbdet(l_{j-1}, o_{i}, l_{j})\}, & \text{otherwise}
\end{cases}
$$

Finally, $LB_{\Delta^*}$ can be calculated as follows.

\[ \text{Algorithm 4: Decision Algorithm} \]

**input**: $\alpha$, workers $W$ and a request $r$  
**output**: a set of lower bound $LB$ for each $w$

1. $LB \leftarrow \{\text{dis}(o_{i}, c_{r}), LB \leftarrow \emptyset$;
2. **foreach** $w \in W$ do
3.  \[ LB_{\Delta^*} \leftarrow \text{Linear DP Insertion}(w, r) \] by using Euclidean distance and $LB$ according to Lemma 7 and Algo. 3;
4. \[ LB \leftarrow LB \cup \{(LB_{\Delta^*}, w)\}; \]
5. **if** $pr < \alpha \cdot \min LB$ then **reject** $r$;
6. **return** $LB$;

5.1.2 Algorithm Sketch

Algo. 4 illustrates the process of the decision process. For each worker $W$, we calculate $LB_{\Delta^*}$ using Eq. (17). Note that we use $LB$ to store all $LB_{\Delta^*}$ for each worker since we will use $LB_{\Delta^*}$ 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 $LB_{\Delta^*}$ in $LB$, which is calculated in the decision phase. If $w_a$ is ahead of $w_b$ in the sorted order and $\Delta^*$ of $w_a$ is smaller than $LB_{\Delta^*}$ of $w_b$, we can safely ignore all workers after $w_a$.

**Proof.** $\Delta^*$ is the actual increased distance of worker $w_a$ and $LB_{\Delta^*}$ is the lower bound of the actual increased distance of worker $w_b$. Since the workers are already sorted according to $LB_{\Delta^*}$, $\Delta^*$ of $w_a$ is also smaller than $LB_{\Delta^*}$ 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 ∅;
2. foreach new request r ∈ R do
   /* Phase 1: Decision */
   3. Cand ← filter the candidate according to grid index, deadline, etc.;
   4. LB ← Decision(α, Cand, r);
   /* Phase 2: Planning */
   5. if r is decided to be served then
      6. w’ ← NIL, ∆∗′ ← ∞;
      7. foreach (LBΔ, w) in sorted LB do
         8. if ∆∗′ < LBΔ then break;
         9. ∆∗ ← Linear DP Insertion(w, r);
         10. if ∆∗ < ∆∗′ then w’ ← w, ∆∗′ ← ∆∗;
      11. if w’ is not NIL then update S∗,w and arr of w’ accordingly;
      12. if r is rejected then R’ ← R’ \ {r};
   13. UC ← α \sum_{w∈W} D(S_w) + \sum_{r∈R−p_r} euc(r);
14. return \{S∗,w\} and UC;

performed using the DP insertion. We use auxiliary array arr[.] 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[.], 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 ∆ based on arr[.]. Beyond the shortest distance query for \( L = \text{dis}(o, d_r) \), we only need the shortest distance between \( o_r, \{d_r\} \) and \( l_1, \ldots, l_n \). Thus, it is 2n + 1 times in total.

### 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 ∆∗ for each w in line 9 and update the currently best worker w’ with minimal increased distance (∆∗) 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[.]. Finally, we calculate the unified cost in line 13.

**Example 3.** Back to the settings in Example 1. Suppose that w1 is assigned to serve r1 at time 0 following route S∗,w1 = \{r7(location of w1), v2, v4\} and w2 is assigned to serve r2 at time 5 following route S∗,w2 = \{r3(location of w2), v3, v5\}.

At time 11, both w1 and w2 travel to v5 when r3 is released. In line 3 of Algo. 5, Cand = \{w1, w2\}, 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 \( L = 4 \) only once and use it to calculate LBΔ with Euclidean distance of coordinates. According to Lemma 7, LBΔ∗ of w1 is euc(v5, v8) + LBΔ∗ of w2 is euc(v8, v5) + L + ecc(v5, v8) − \{arr_w1[2] − arr_w2[1]\} = 0 + 4 + 2.8 − 5 = 1.8.

### 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 GAAI 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 location, etc.; which is the input \( \{w, r\} \). The second is a public dataset [8] that is released.

**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., \( O(|R|) \)). 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 \( O(|R|^2 q + |R||W| q + |R||W| \log W) \).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#(Requests)</th>
<th>#(Vertices)</th>
<th>#(Edges)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NYC</td>
<td>917,100</td>
<td>807,795</td>
<td>2,100,632</td>
</tr>
<tr>
<td>Chengdu</td>
<td>259,347</td>
<td>214,440</td>
<td>466,330</td>
</tr>
</tbody>
</table>

**Table 4: Statistics of datasets.**
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 \( t_f = t_r + 10 \text{ min} \). \( K_w \) is generated using a gaussian distribution with \( \mu = 3, \cdots, 20 \), because neither dataset specifies this information. We fix \( \alpha \) 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., \( p_r = 10 \times \text{dis}(o_i, d_i) \) by default. Note that \( \alpha \) is fixed to 1 and \( p_r = 2, \cdots, 50(\times \text{dis}(o_i, d_i)) \), which is equivalent to adjusting the proportion between \( c_r \) and \( e_r \) 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 \texttt{pruneGreedyDP} with the following state-of-the-art algorithms.

- \texttt{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.

- \texttt{kinetic} [25]. It uses a kinetic tree to maintain every possible route to serve all the remaining requests. Unlike \texttt{tshare}, the insertion operation is recursively executed based on the tree structure.

- \texttt{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.

- \texttt{GreedyDP}. It is a variant of our \texttt{pruneGreedyDP} algorithm without the pruning strategy in Lemma 8.

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 \texttt{pruneGreedyDP} and \texttt{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, \texttt{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, \texttt{pruneGreedyDP} has the highest served rate, 54.94% and 141.61% higher than \texttt{batch}. The results of served rate in \( Chengdu \) indicate that \texttt{pruneGreedyDP} is competitive with \texttt{kinetic} and better than \texttt{batch} when maximizing the number of served requests. For a larger road network, the
<table>
<thead>
<tr>
<th>K_w</th>
<th>Unified Cost</th>
<th>Served Rate</th>
<th>Response Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Chengu</td>
<td>0.9</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>NYC</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>6</td>
<td>_batch</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>10</td>
<td>_pruneGreedyDP</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>20</td>
<td>_batch</td>
<td>1.1</td>
<td>0.4</td>
</tr>
</tbody>
</table>

**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 (2^K_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 _ε_r_.** Fig. 6 shows the results of varying the deadline _ε_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 _ε_r_ = 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 _ε_r_ increases from 5 to 25 minutes. Conversely, the response time of _pruneGreedyDP_ remains within 50ms. This is because when varying _ε_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_.** 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-
Served Rate | Response Time (secs) | Response Time (secs) | Unified Cost |
---|---|---|---|
0.6 | 0.1 | 0.2 | 0.2
0.1 | 0.1 | 0.2 | 0.5
0.2 | 0.2 | 0.5 | 0.8
0.2 | 0.2 | 0.2 | 0.2
0.8 | 0.4 | 0.4 | 1.5
0.4 | 0.4 | 0.3 | 0.3
0.3 | 0.3 | 0.3 | 0.3
0.4 | 0.4 | 0.4 | 1.6
1.5 | 0.4 | 0.5 | 0.8
0.4 | 0.5 | 0.6 | 1.4
0.6 | 0.6 | 0.6 | 1.6

\[ \text{Figure 6: Performance of varying deadline } e_r. \]

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

- Our \textit{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, \textit{GreedyDP} and \textit{pruneGreedyDP}, are 2.62 to 20.72 times faster than the state-of-the-arts [25][11]. In many cases, \textit{pruneGreedyDP} is 2.8 times faster than \textit{GreedyDP}, due to tens of billions of shortest distance queries saved.

- Among the state-of-the-arts, \textit{kinetic} [25] often fails to halt in time on large-scale datasets for its exponential time complexity. \textit{batch} [11] is less effective and efficient than our solution but more scalable than \textit{kinetic} in large-scale datasets. \textit{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 quadratic 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 Collaborative 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. 61722001, the Science and Technology Planning Project of Guangdong Province, China, No. 2015B010100006, Microsoft Research Asia Collaborative Research and HKUST SSTP under Project FP305.
REFERENCES


