Real-time Insertion Operator for Shared Mobility on Time-Dependent Road Networks

Zengyang Gong[†] Yuxiang Zeng[*] Lei Chen[†]

[†]The Hong Kong University of Science and Technology, Hong Kong SAR, China

[*]School of Computer Science and Engineering, Beihang University, Beijing, China

{zgongae, leichen}@cse.ust.hk, yxzeng@buaa.edu.cn

ABSTRACT

One of the most important challenges in shared mobility services (*e.g.*, ride-sharing and parcel delivery) is planning routes for workers by considering real road conditions. To tackle this challenge, the "insertion operator", which computes the optimal route for the worker to serve (*i.e.*, insert) the newly appeared delivery request, has been acted as the fundamental operation in existing solutions. However, existing work implicitly assumed a static road network, and hence was hard to fulfill the real-world scenario, where travel time between two locations is not constant at different times of a day. By contrast, we focus on the insertion operator over time-dependent road networks that capture the periodic pattern of road conditions. We also show that the time complexity of existing solutions would degrade into cubic time and hence such solutions can no longer satisfy the real-time requirement under this real-world setting. To satisfy the need for real-time computation, we propose a data summary to model the time-dependent travel time functions between pairs of vertices in the route. Based on the data summary, we design an efficient solution that can enumerate the best insertion position in linear time while satisfying complex spatiotemporal constraints. Finally, extensive experiments are conducted on real datasets from several applications of shared mobility. The results show that our solution is up to 44.5× faster than the state-of-the-art solution.

PVLDB Reference Format:

Zengyang Gong, Yuxiang Zeng, and Lei Chen. Real-time Insertion Operator for Shared Mobility on Time-Dependent Road Networks. PVLDB, 17(1): XXX-XXX, 2024.

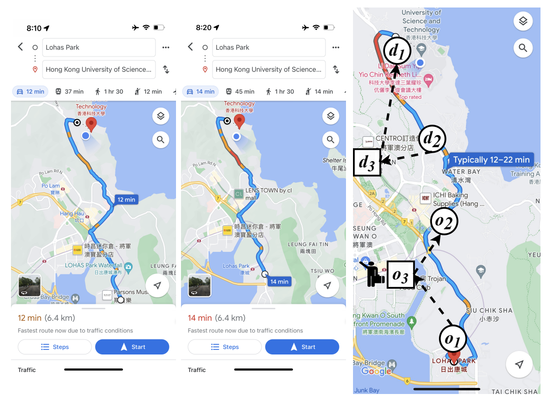
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/gzyhkust/Insertion-Operator.git.

1 INTRODUCTION

Shared mobility services [33], such as ride-sharing and food delivery, allow a worker to provide the shared ride for multiple requests with similar traveling schedules. For example, in a ride-sharing platform (*e.g.*, DiDi Chuxing [2]) or a food delivery platform (*e.g.*, Meituan [4]), a worker (*e.g.*, a driver or courier) can carry more than one passenger or food parcel that have similar origins and destinations. Finding a proper and practical route online for the worker to serve a newly appeared request is one of the most significant challenges in these platforms, especially during peak hours when the number of requests is large-scale.

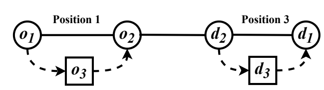


(a) Real-world road networks. **(b) Insert request r_3.**

Figure 1: Insert a new request on real-world road networks.

Position pairs for inserting o_3 and d_3			
(1, 1)	×	×	×
(1, 2)	(2, 2)	×	×
(1, 3)	(2, 3)	(3, 3)	×
(1, 4)	(2, 4)	(3, 4)	(4, 4)

(a) All possible position pairs.

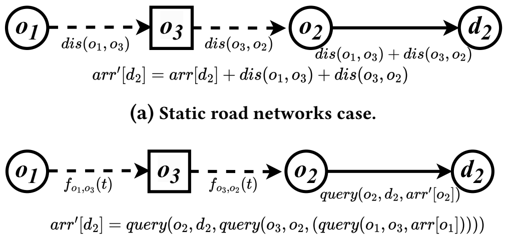


(b) Insertion pair $(1, 3)$.

Figure 2: Insertion operator.

In the past years, planning routes in shared mobility has been widely studied [7, 11, 12, 18, 21, 22, 25, 31–33, 35, 40, 41, 43–45]. Most of these solutions were built upon a frequently-used operator called *insertion*. This operator aims to find the optimal route with minimum increased travel time to serve (or attempt to serve) a newly appeared request for a worker on the real-world road network, ensuring that the route is identified immediately when the request appears on the platform and that the worker can deliver the request before its deadline. In real industry (*e.g.*, [28, 40]) more than one worker is on the way to serve the new request, and the one, who takes the minimum increased travel time, will be allocated to the request. When no worker is close enough, the request may need to wait for some time until being served or canceled. Prior studies [34, 38] have demonstrated that the insertion operator with low time complexity (*e.g.*, linear time) can reduce the efficiency bottleneck.

Proceedings of the VLDB Endowment, Vol. 17, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX



(b) Real road networks case.

Figure 3: Calculate the new arrival time at d_2.

Motivation. Despite extensive research on optimizing the insertion operator, existing solutions still remain a challenging obstacle to be deployed in real-world applications, *i.e.*, a common and implicit assumption on the static road network. In other words, these solutions all failed to consider the dynamic nature of road conditions that are affected by weather, traffic congestion, etc. Unfortunately, when considering this real-world factor, we find that their solutions will either produce infeasible routes or significantly increase the time complexity. Here, we take an example to explain this finding as follows.

Motivation Example. Fig. 1a shows an example that the shortest travel time between any two locations is not constant on a real-world road network. For instance, Google Map [3] shows the travel time of a single route at different times in the morning, *i.e.*, 12 minutes at 8:10 am and 14 minutes at 8:20 am. This delay is due to the peak hours of the daily traffic. In fact, the travel time of this route periodically varies between 12 and 22 minutes every day. Thus, planning a practical route in shared mobility needs to consider the dynamic nature of road networks.

Due to this reason, it is non-trivial to study the insertion operator over time-dependent road networks that are commonly used to model real-world city networks [8, 36, 37]. Fig. 1b shows an example of inserting a new request on a real-world road network. Suppose a taxi driver is traveling from the location o_1 to the location d_1 to serve two passengers, where o_1-o_2 are the passengers' origin and d_1-d_2 are the passengers' destination. At some point, the third passenger r_3 submits his ride-sharing request to the platform, and the insertion operation is used to compute the minimum increased travel time for this worker to additionally serve the newly appeared request with the origin o_3 and the destination d_3.

The insertion operation considers $O(n^2)$ pairs of positions to insert the new request's origin and destination into the current route, where n is the number of locations in the current route. The value of n can be more than 30 in real-world applications according to a recent statistic report [6]. One possible pair $(1, 3)$, which inserts o_3 between o_1 and o_2 and d_3 between d_2 and d_1 and causes two detours marked by dashed arrows, is as shown in Fig. 2b. For this case, existing solutions calculate the increased travel time by adding up the travel time of these detours. *Although such an additivity property can be established in a static road network, it no longer holds in a real-world road network.* For instance, after inserting o_3 at position 1, directly adding up the delay time (*i.e.*, $dis(o_1, o_3) + dis(o_2, o_1)$ at 8:10 am, $dis()$ is the static shortest travel time) of detours to get the new arrival time at d_2, while sequentially querying the delay time at each location based on the new route is $query(o_2, d_2, query(o_3, o_2, (query(o_1, o_3, arr[o_1]))))$ (the function $query()$ is the shortest arrival time query over real road networks, see Fig. 3 for details). Thus, the additivity property leads to an inaccurate travel time, which could delay the passenger's original trip and significantly hurt user experiences. Similar counter-examples can be easily found in real life.

Table 1: Summary of major notations.

Notation	Description
$G(V, E, F)$	time-dependent road network
$f_{i,j}(t)$	$f_{i,j} \in F$ weight function of edge $e_{i,j}$;
$f'_{x,y}(t)$	weight function of the route from v_x to v_y
n	number of locations a worker visits to serve assigned requests
$query(u, v, t)$	shortest arrival time query over $G(V, E, F)$
$ComTravel(u, v, t)$	compound travel function over route u to v
v_w, c_w	location and capacity of worker w
o_r, d_r	origin and destination of request r
$pik(o_r), del(d_r)$	pickup time and deliver time of request r
$arr[v_k]$	arrival time at v_k along the route of worker
$num[v_k]$	number of picked request at v_k along the route of worker
$latest[v_k]$	the latest arrival time at v_k to keep feasibility of the route
$opt_i[v_k]$	the optimal i value when insert d_r before v_k

Our Solution Summary. Motivated by this limitation, we study the insertion operator for route planning in shared mobility on time-dependent road networks that are commonly used to model the dynamic nature of road conditions (see Sec. 2.1 for more details). To meet the optimality and real-time requirements, we first observe that the key point to hinder the efficiency is how to efficiently answer a spatiotemporal query, *i.e.*, delay time query ($delayquery()$). To rapidly process such a query, we design a data summary model called compound travel functions to capture the travel time between locations in the route. By utilizing this data structure, we propose an efficient solution to improve the time complexity from $O(n^3)$ to $O(n)$.

Contribution. The main contributions are listed as follows:

- We are the first to study the insertion operation for route planning in shared mobility on time-dependent road networks.
- We identify that existing solutions to the insertion operations over static road networks cannot always retain the optimal result under the setting of road networks. In our experiments, it could delay 27% requests' trips.
- To address this limitation, we design an efficient solution by utilizing a novel data summary model. We also prove that our solution only takes linear time to compute the exact result, which is no higher than that of existing work.
- Extensive experiments on real datasets from various applications demonstrate that our solution can accelerate the insertion operation by up to 44.5× in the running time.

Roadmap. The rest of this paper is organized as follows. We first present the problem statement in Sec. 2. Then, our method is elaborated in Sec. 3 and evaluated in Sec. 4. Finally, we review the related work in Sec. 5 and conclude in Section Sec. 6.

2 PROBLEM STATEMENT AND BASELINE METHOD

This section introduces the basic concepts (Sec. 2.1), formal problem definition (Sec. 2.2), and exact baseline method (Sec. 2.3). The major notations are summarized in Table 1.

2.1 Basic Concepts

Definition 1 (Time-dependent Road Network)**.** A directed graph *G(V, E, F)* is used to model the real-world road network. Here, V is the set of vertices and each vertex $v \in V$ represents one geo-location. $E \subseteq V \times V$ is the set of edges. Each directed edge $(u, v) \in E$ is associated with a non-negative weight function $f_{u,v}(t) \in F$, where t is the departure time, and $f_{u,v}(t)$ denotes the travel time to the vertex v when departing from the vertex u at time t. Besides, for any two vertices $u, v \in V$, the function $query(u, v, t)$ denotes the shortest arrival time to v when departing from u at time t.

Following the conventions in Ref. [8, 36, 37], we adopt piecewise linear functions (PLF) to model the weight functions in real road networks, any continuous function of the weight edges can be approximated by a set of such functions [9]. Specifically, a weight function $f_{u,v}(t)$ associated with each directed edge $(u, v) \in E$ is modelled as a set of interpolation points $P = \{(t_1, w_1), (t_2, w_2), \ldots, (t_k, w_k)\}$, and each point (t_i, w_i) denotes that it takes w_i unit time to travel from u to v at time t_i. Besides, a straight line connected two successive points $(t_i, w_i), (t_{i+1}, w_{i+1})$ fits the linear function in the time domain $[t_i, t_{i+1})$. For example, if a worker departs from u at time $t \in [t_1, t_2)$, his travel time can be calculated as $f_{u,v}(t) = w_1 + (t - t_1)\frac{w_2 - w_1}{t_2 - t_1}$. Then, the weight function of (u, v) can be formalized by Eq. (1).

$$f_{u,v}(t) = \begin{cases} w_1 + (t - t_1)\frac{w_2 - w_1}{t_2 - t_1}, & t_1 \le t < t_2 \\ w_2 + (t - t_2)\frac{w_3 - w_2}{t_3 - t_2}, & t_2 \le t < t_3 \\ \cdots & \\ w_{k-1} + (t - t_{k-1})\frac{w_k - w_{k-1}}{t_k - t_{k-1}}, & t_{k-1} \le t \le t_k \end{cases} \tag{1}$$

where the time domain of the function $f_{u,v}(t)$ is $[t_1, t_k]$.

Property of Time-dependent Road Networks. In the real world, if two people move from u to v at different times, the one who departs earlier usually arrives at v no later than the other one. This phenomenon is summarized as one important property by existing work [8, 36, 37], *i.e.*, the first-in-first-out (FIFO) property. This property implies $t_1 + f_{u,v}(t_1) \le t_2 + f_{u,v}(t_2)$ for any weight function $f_{u,v}(t) \in F$ and departure time $t_1 \le t_2$.

Based on Definition 1 and the conventions of existing work [17, 20, 26, 33, 34], we present the definitions of other basic concepts, including a request and a worker, in Definition 2 and 3.

Definition 2 (Request)**.** A request is denoted by $r = \langle o_r, d_r, t_r, e_r, c_r \rangle$. This request appears on the platform at time t_r. $o_r \in V$ is the request's origin, $d_r \in V$ is the destination, and e_r is the deadline time. The size c_r denotes the number of passengers or parcels that need to be carried.

In a real-world platform, a request is *said to be served* if it is picked up at the origin o_r by a worker with enough capacity (compared with the size c_r) after the appearance time t_r and delivered to the destination d_r before the deadline time e_r. A worker is formulated as follows.

Definition 3 (Worker)**.** A worker is denoted by $w = \langle c_w, \mathcal{S}_\mathcal{R}, t_0 \rangle$, where R is the set of requests that have been allocated to this worker but undelivered, c_w is the worker's capacity, and $\mathcal{S}_\mathcal{R}$ is his current route at the time t_0. Here, the route is defined as a sequence of vertices in V, i.e., $\mathcal{S}_\mathcal{R} = \langle v_0, v_1, \cdots, v_n \rangle$, where v_0 is his current location, and $v_1, \cdots, v_n$ is either an origin or a destination of an undelivered request in R. Besides, we use $arr[v_k]$ to denote the arrival time at the location v_k by following this route, i.e.,

$$arr[v_k] = \begin{cases} t_0, & k = 0 \\ query(v_{k-1}, v_k, arr[v_{k-1}]), & k > 0 \end{cases}$$

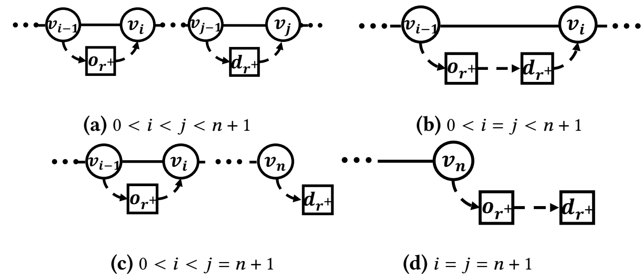


Figure 4: Insertion cases of all possible pairs(i, j).

In Definition 3, the worker's capacity implies that the total size of the requests that are carried by him at any time cannot exceed c_w. For instance, c_w is often 3-4 in ride-sharing [15, 25, 40], while could be 80-200 in food/parcel delivery [23, 30, 38, 46]. According to the above definition, we can easily compute the pickup time and delivery time of a request, which are denoted by $pik(o_r)$ and $del(d_r)$, respectively. In a shared mobility platform, a route is *said to be feasible* if all the requests assigned to this worker are delivered before the deadline.

2.2 Problem Definition

According to the above concepts and conventional definition [34, 38] of the insertion operation, we present the formal definition of the *insertion operator for shared mobility on time-dependent road networks* ("*time-dependent insertion*" as short).

Definition 4 (Time-dependent Insertion)**.** Given a worker w and a request r^+ that newly appears at time t_{r^+}, this operation finds a new route for this worker S^* with the minimum increased travel time obj^* in real-time to serve all the requests R^+ (i.e., $R^+ = R \cup r^+$) while satisfying the following constraints:

- **Completion Constraint.** All the requests must be served.
- **Order Constraint.** The relative orders of vertices in S_R remain the same in S^*. In other words, the new request's origin and destination are sequentially inserted at some positions of the worker's current route.
- **Deadline Constraint.** All the requests must be delivered to its destination by this worker before the deadline time.
- **Capacity Constraint.** At any time, the total size of requests that this worker has picked up but not delivered is no larger than the worker's capacity.

The insertion operator $insert(i, j)$ indicates that the new request's origin/destination is inserted at the i/j-th position of the current route $\mathcal{S}_\mathcal{R}$ (*i.e.*, before the vertex v_i/v_j), where $1 \le i \le j \le n+1$. Fig. 4 shows all four cases of the new route after inserting the new request.

2.3 Baseline Method

An extension of an existing $O(n^3)$-time method for static road networks [38] serves as our baseline for developing an insertion operator on time-dependent road networks. We present the implementation details of this method in Algo. 1.

Main Idea. The intuition behind the baseline method is as follows. Specifically, we enumerate all pairs of potential positions (i, j) to insert the new request's origin o_{r^+} and new request's destination d_{r^+} and obtain a possible route S_{R^+} (lines 2-4). Based on this possible route, we compute the new arrival time at each vertex and check the feasibility (lines 5). If there is no constraint violation and the increased travel time of this route is shorter than that of the currently optimal route S^*, we will replace S^* with S_{R^+} (lines 6-8).

Complexity Analysis. Suppose the time cost of the shortest arrival time query on a time-dependent road network is $O(q)$. Lines 2-3 take $O(n^2)$ time. Lines 5-6 take $O(nq)$ time. Thus, the overall time complexity is $O(n^3q)$ time. For brevity, existing work usually ignores the term $O(q)$, so the baseline takes cubic time. We will also follow this convention in the rest of the paper.

Algorithm 1: Baseline: Cubic Time Insertion [38]

```
   Input: a worker w and a new request r+
   Output: route S* with min increased travel time obj*
1  S* ← S_R, obj* ← +∞, arr[v_0] ← t_0 ;
2  for i ← 1 to n + 1 do
3      for j ← i to n + 1 do
4          S_R+ ← insert new request's o_r+/d_r+ before the
             vertex v_i/v_j in S_R ;
5          if S_R+ satisfies all constraints in Definition 4 then
6              obj ← increased travel time of S_R+;
7              if obj < obj* then
8                  S* ← S_R+, obj* ← obj;
9  return S*, obj*;
```

3 OUR METHODOLOGY

To address the limitations of the baseline method, we introduce a data summary model called the compound travel functions that accelerates the delay time query along the worker's route. This data structure enables efficient computation of the increased travel time and feasibility checking in $O(1)$ time. This directly reduces the time complexity of a time-dependent insertion to $O(n^2)$ by consuming $O(n^2)$ space cost. After that, we show how to find the optimal i in $O(1)$ time when j is given along a feasible route of the worker by solely utilizing the compound travel function from v_{j-1} to v_j. As a result, both time and space complexity can be further improved to $O(n)$ by enumerating j along the worker's feasible route.

3.1 Preliminary

3.1.1 *Data Summary Model: Compound Travel Functions.*

To accelerate the increased travel time computation and feasibility checking, we calculate one $ComTravel(v_x, v_y, t)$ in Definition 5 for each pair of vertices (v_x, v_y) in the feasible route $S_R = \langle v_0, v_1, \dots v_n \rangle$ where $0 < x < y < n$. This function represents the travel time when starting from vertex v_x at time t and ending at vertex v_y.

We refer to the compound travel functions maintained for a worker's current route as the data summary model. The formal definition of compound travel functions along a route is given as:

Definition 5 (Compound Travel Function). *Given two vertexes v_x and v_y associated with the sub-route $\langle v_x, v_{x+1}, \cdots, v_y \rangle$ in S_R, the compound travel function $ComTravel(v_x, v_y, t)$ indicates the travel time of the route when start from v_x to v_y at time t, $ComTravel(v_x, v_y, t)$ $= f'_{y-1,y}(f'_{y-2,y-1}(\cdots f'_{x,x+1}(t))) - t$.*

For a successive sub-route $\langle v_k, v_{k+1} \rangle \in S_R (0 \le k \le n-1)$, we assume that there is a shortest path $(u_0 = v_k, u_1, u_2 = v_{k+1})$ in the graph connecting v_k and v_{k+1}, with (u_0, u_1) and (u_1, u_2) being edges of the graph. We can obtain two edge weight functions $f_{0,1}(t)$ and $f_{1,2}(t)$ associated with these two edges. Starting from u_0 at time t_0, we can compound t_0 to the edge weight function $f_{0,1}(t)$ to obtain the arrival time at u_1, represented as a function $f_{0,1}(t_0)$. We can then compound $f_{0,1}(t_0)$ into the edge weight function $f_{1,2}(t)$, which gives the function $f_{1,2}(f_{0,1}(t_0))$ representing the arrival time at u_2 when starting from v_0 at t_0. Finally, we can use the $f'_{k,k+1}(t_0) =$ $f_{1,2}(f_{0,1}(t_0))$ to denote the weight function along the successive sub-route. It can be easily calculated by the existing shortest path query algorithm [36] on time-dependent road networks.

Example 2. *As the sub-route $\langle o_1, o_2, d_2 \rangle$ shown in Fig. 1. If the weight functions for $\langle o_1, o_2 \rangle$ and $\langle o_2, d_2 \rangle$ are $w_{o_1,o_2}(t) = \{(0, 10), (60, 20)\}$ and $w_{o_2,d_2}(t) = \{(0, 5), (60, 30)\}$, respectively. At time 0, the travel cost from o_1 to d_2 is calculated as $10 + (5 + 10 * \frac{30-5}{60-0})$, where 10 is the cost of $\langle o_1, o_2 \rangle$ at time 0, and $(5 + 10 * \frac{30-5}{60-0})$ denotes the cost of $\langle o_2, d_2 \rangle$ when starting the edge at time 10. Using this method, we can calculate the compound travel function $ComTravel(o_1, d_2, *)$.*

3.1.2 *Query the Delay Time by Compound Travel Function.*

The compound travel function accelerates the travel time computation to $O(1)$ time, based on this data structure, we design a novel spatiotemporal query delay time query along the worker's route.

Definition 6 (Delay time query). *Given two vertexes v_x and v_y in worker's route, where v_y locates after v_x, the delay time query $delayquery(v_x, v_y, t)$ retrieves the delay time at v_y when the arrival time v_x is delayed to time t, where $delayquery(v_x, v_y, t) = arr[v_x] +$ $ComTravel(v_x, v_y, t) - arr[v_y]$.*

After inserting the new origin o_{r^+}/destination d_{r^+} before v_i/v_j in S_R. The new arrival time at v_i and v_j are dependent on the pickup time and delivery time of r^+. For vertices in the sub-route $\langle v_{i+1}, v_{i+2}, \dots, v_{j-1} \rangle$, which locates after v_i and before v_j, the delay time at these locations are dependent on the new arrival time at v_i, which can be calculated by the delay queries $delayquery(v_i, \cdot, \cdot)$. The sub-route $\langle v_{j+1}, v_{j+2}, \cdots, v_n \rangle$ consists of the remaining vertices in S_R after v_j, and the delay time of these vertices are dependent on the arrival time to v_j, which can be calculated by the delay queries $delayquery(v_j, \cdot, \cdot)$. We next show how to calculate the arrival time of each vertex in S_{R^+}, based on its original position in S_R.

- **Computing for pickup time:** New origin o_{r^+} is inserted before v_i and after v_{i-1}, so the pickup time can be calculated from arrival time to v_{i-1} in the original route.

$$pik(o_{r^+}) = query(v_{i-1}, o_{r^+}, arr[v_{i-1}]) \tag{2}$$

- **Delay time query in $\langle v_i, v_{i+1}, \cdots, v_{j-1} \rangle$:** The insertion of the new origin o_{r^+} before v_i caused the delay at vertices in this sub-route, which needs to be calculated. The new arrival time at v_i is dependent on the pickup time, $arr'[v_i] =$

$query(o_{r^+}, v_i, pik(o_{r^+}))$. For other vertices, the delay time caused by new arrival time $arr'[v_i]$ can be calculated as $arr[v_k] + delayquery(v_i, v_k, arr'[v_i]), i < k < j$.

- **Computing for delivery time:** We consider two cases. If $i < j$, d_{r^+} is inserted after v_{j-1} and before v_j, the delivery time can be calculated from the new arrival time to v_{j-1} (as shown in Figure 4a and 4c). On the other hand, if $i = j$, d_{r^+} is inserted immediately after o_{r^+}, so the delivery time can be calculated from the pickup time of the new request (as shown in Figure 4b and 4d).

$$del(d_{r^+}) = \begin{cases} query(v_{j-1}, d_{r^+}, arr'[v_{j-1}]), & i < j \\ query(o_{r^+}, d_{r^+}, pik(o_{r^+})), & i = j \end{cases} \tag{3}$$

- **Delay time query in $\langle v_j, v_{j+1}, \cdots, v_n \rangle$.** After inserting d_{r^+}, the new arrival time at v_j is dependent on the delivery time, $arr'[v_j] = query(d_{r^+}, v_j, del(d_{r^+}))$. For other vertices, the delay time caused by the new arrival time at v_j can be calculated as $arr[v_k] + delayquery(v_j, v_k, arr'[v_j]), j < k \leq n$

Based on the delay time query in Definition 6, the time complexity of the calculations for each possible pair (i, j) is $O(1)$.

3.2 Improve Time Complexity to $O(n^2)$ by using $O(n^2)$ Compound Travel Functions

Basic Idea. To accelerate the constraints checking and increased travel time computation from $O(n)$ to $O(1)$ in the baseline, a compound travel function is built for each insertion positions pair (i, j). Therefore, a total of $O(n^2)$ compound travel functions are maintained. These functions compute the delay time in each vertex after insertion in $O(1)$ time, enabling efficient constraint checking and calculation of the increased travel time.

3.2.1 *Checking Constrains by using $O(n^2)$ Compound Travel Functions*.

Given a possible (i, j) pair for insertion, the feasibility of the new route must be verified against two constraints: the deadline constraint and the capacity constraint.

Checking Deadline Constrain. As discussed in Sec. 3.1.2, we first use Eq. (4) to calculate the arrival time of each vertex in S_{R^+} in $O(1)$ time. For vertexes v_i and v_j, the new arrival times are dependent on the pickup time and delivery time of the new request, two *query* functions are invoked to calculate the new arrival time to them respectively. For vertexes locate between v_{i+1} and v_{j-1}, $delayquery(v_i, *, *)$ query the delay time at these vertices. Similarly, the delay time queries $delayquery(v_j, *, *)$ can be utilized to calculate the delay times of vertexes located after v_j. Specially, the time complexity of the shortest arrival time query and the delay time query based on compound travel functions are both $O(1)$. Therefore, for each possible insertion position pair (i, j), we can calculate the arrival time of each vertex belonging to S_{R^+} in $O(1)$ time.

$$arr'[v_k] = \begin{cases} arr[v_k], & k < i \\ query(o_{r^+}, v_i, pik(o_{r^+})), & k = i \\ arr[v_k] + delayquery(v_i, v_k, arr'[v_i]), & i < k < j \\ query(d_{r^+}, v_j, del(d_{r^+})), & k = j \\ arr[v_k] + delayquery(v_j, v_k, arr'[v_j]), & j < k \leq n \end{cases} \tag{4}$$

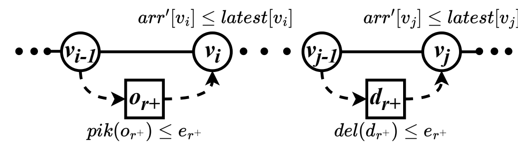


Figure 5: Checking deadline constraint.

We further use $latest[v_k]$ to denote the latest arrival time at vertex v_k without violating any deadline constraints at v_k and all vertices after v_k. Similar to calculating the arrival time at a vertex, $latest[v_k]$ can be computed from the compound travel function.

To ensure that the worker satisfies the deadline constraint for all vertices from v_k to v_n, we need to calculate $latest[v_k]$ in reverse order for each successive sub-route $\langle v_k, v_{k+1} \rangle \in S_R$, where k ranges from n to 0. The latest arrival time for each vertex can be calculated using the following equation, where $latest[v_n]$ is the deadline time of the request whose destination is the last vertex v_n:

$$\begin{cases} latest[v_k] + ComTravel(v_k, v_{k+1}, latest[v_k]) = latest[v_{k+1}], \\ latest[v_n] = e_r, \text{ when } v_n = d_r \end{cases} \tag{5}$$

The following lemmas check the deadline constraint.

Lemma 1. The deadline constraint will not be violated if and only if (1) $pik(o_{r^+}) \leq e_{r^+}$; (2) $arr'[v_i] = query(o_{r^+}, v_i, pik(o_{r^+})) \leq latest[v_i]$; (3) $del(d_{r^+}) \leq e_{r^+}$; (4) $arr'[v_j] = query(d_{r^+}, v_j, del(d_{r^+})) \leq latest[v_j]$;

Proof. We prove this lemma by considering the general insertion case, as depicted in Fig. 5. Given a new request r^+, we first calculate the pickup and delivery time in conditions (1) and (3), respectively. These two conditions ensure that the new potential route will not violate the deadline constraint of r^+. Condition (2) calculates the new arrival time at v_i after picking up r^+, denoted as $arr'[v_i]$. To ensure that the deadline constraints after v_i are satisfied, $arr'[v_i]$ cannot exceed the latest arrival time to v_i, which is denoted as $latest[v_i]$. Similarly, for vertex v_j, condition (4) calculates the new arrival time after serving the new request, denoted as $arr'[v_j]$. In order not to violate the deadline constraints for vertices located after v_j, condition (4) guarantees that $arr'[v_j]$ is earlier than the latest arrival time to v_j, which is denoted as $latest[v_j]$. □

Checking Capacity Constrain. To check the capacity constraint in $O(1)$ time, we use $num[v_k]$ to represent the number of requests picked up but not yet delivered at vertex v_k, following previous work [34]. We calculate this array according to the following rules, where v_k is either a pickup location o_r or a delivery location d_r for a request $r \in R$:

$$num[v_k] = \begin{cases} num[v_{k-1}] + c_r, & v_k = o_r \\ num[v_{k-1}] - c_r, & v_k = d_r \end{cases} \tag{6}$$

Lemma 2. [34] Along the new route, the capacity constraint will not be violated if and only if (1) $num[v_{i-1}] \leq c_w - c_{r^+}$; (2) $num[v_k] \leq c_w - c_{r^+}, i < k \leq j$.

Please refer to Ref. [34] for the proof of Lemma 2.

3.2.2 *Computation of Increased Travel Time*.

If the position pair (i, j) satisfies both deadline and capacity constraints, we can call this pair feasible for insertion. Then the increased travel time can be calculated as obj in the following:

$$obj = \begin{cases} delayquery(v_j, v_n, arr'[v_j]), & j \leq n \\ del(d_{r^+}) - arr[v_n], & otherwise \end{cases} \tag{7}$$

When $j \leq n$, the new request is delivered before reaching v_j, resulting in a delay at v_j. By utilizing the compound travel functions maintained for v_j, we can directly perform the delay time query to the last vertex v_n to calculate the increased travel time caused by the delay at v_j in $O(1)$ time. On the other hand, if the new request is the last one to be delivered, as shown in Figure 4c and 4d, the increased travel time is simply the delivery time of the new request minus the arrival time of the last location in the original route.

Algorithm 2: Quadratic Time Insertion

Input: a worker w with a feasible route S_R and a request r^+
Output: new route S^* for w and *min* increased travel time obj^*

1 $S^* \leftarrow S_R, obj^* \leftarrow +\infty$;
2 Initialize *latest*, *num* by *Eq.* (5) - *Eq.* (6) ;
3 Compound travel functions $ComTravel(i, j, \cdot), \forall i, j, 0 < i < j < n$ by Definition 5 ;
4 **for** $i \leftarrow 1$ *to* $n + 1$ **do**
5   **if** *Lemma 1 (1) or (2) violated* **then** continue;
6   **if** *Lemma 2 (1) violated* **then** continue;
7   **for** $j \leftarrow i$ *to* $n + 1$ **do**
8   **if** *Lemma 1 (3) or (4) violated* **then** continue;
9   **if** *Lemma 2 (2) violated* **then** break;
10         $obj \leftarrow Eq.\ (7)$;
11   **if** $obj < obj^*$ **then**
12             $obj^* \leftarrow obj, i^* \leftarrow i, j^* \leftarrow j$;
13 **if** $obj^* < +\infty$ **then**
14     $S^* \leftarrow$ insert o_r at i^*-th and d_r at j^*-th in S_R ;
15   Get new arrival time of S^* by *Eq.* (4) ;
16 **return** S^*, obj^*;

3.2.3 *Algorithm Details.*

Main Idea. We enumerate all possible (i, j) pairs for insertion in the algorithm, and build one compound travel function for each pair. With the total of $O(n^2)$ functions, the constraints checking and increased time computation can be achieved in $O(1)$ time. The new arrival times and increased travel time are updated by Eq. (4) and Eq. (7) in $O(1)$ time respectively. The deadline constraint is checked for v_i using Lemma 1 (1)-(2) after inserting o_{r^+}, and the capacity constraint is checked using Lemma 2 (1). For v_j, the algorithm uses Lemma 1 (3)-(4) and Lemma 2 (2) to check the deadline and capacity constraints after inserting d_{r^+} before it. All these feasibility checks have a time cost of $O(1)$.

Complexity Analysis. Algo. 2 shows the detail. Line 2 initializes auxiliary arrays *latest*, *num*. Line 3 computes the compound travel function between any two vertexes along S_R to build the data summary model. Lines 4-9 enumerate possible positions to insert o_{r^+} and d_{r^+}, checking the deadline and capacity constraints in $O(1)$ time for each position. If (i, j) is feasible, the increased travel time is calculated in line 10 in $O(1)$ time. Therefore, the total time complexity of Algo. 2 is $O(n^2)$, and the space complexity is also $O(n^2)$, as the worker maintains $O(n^2)$ compound travel functions.

Table 2: $ComTravel$ **of** S_R **in Algo. 2**

	o_1	o_2	d_2	d_1
o_1	×	$ComTravel(o_1, o_2, t)$	$ComTravel(o_1, d_2, t)$	$ComTravel(o_1, d_1, t)$
o_2	×	×	$ComTravel(o_2, d_2, t)$	$ComTravel(o_2, d_1, t)$
d_2	×	×	×	$ComTravel(d_2, d_1, t)$
d_1	×	×	×	×

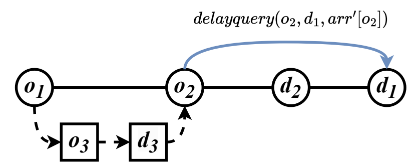


Figure 6: Update arrival time with $ComTravel$**.**

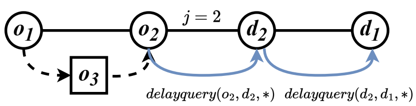
Example 3. *Back to the settings in Fig. 1. For the current feasible route* $S_R = \langle o_1, o_2, d_2, d_1 \rangle$*, the compound travel functions along* S_R *are shown in Table 2. The algorithm computes (3+2+1) compound travel functions. For each vertex in* S_R*, the algorithm computes and maintains compound travel functions to all vertices after it. To serve* r_3*, we consider the first possible insertion pair* $(1, 1)$*. As shown in Figure 6, by inserting* o_3 *and* d_3 *before* o_2*, the resulting increase in travel time can be directly calculated as* $delayquery(o_2, d_1, arr'[o_2])$ *in* $O(1)$ *time. This delay time query is facilitated by the compound travel function* $ComTravel(o_2, d_1, arr'[o_2])$*, which is maintained by* o_2*. Alternatively, without this function, the sequential invocation of the shortest arrival time query* $query(d_2, d_1, query(o_2, d_2, arr'[o_2])))$ *scans the sub-route* $< o_2, d_2, d_1 >$ *to calculate the increased travel time.*

Correctness. The algorithm can find the optimal position pair (i, j) with the minimum increased travel time to deliver the assigned new request r^+. The order constraint can be trivially verified by the pseudo code of Algo. 2. The feasibility of (i, j) for insertion is ensured by the correctness of Lemma 1 and Lemma 2, while *Eq.* (4) - *Eq.* (7) guarantee the minimum increased travel time.

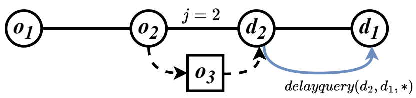
Discussion. This algorithm requires $O(n^2)$ compound travel functions, which results in significant memory overhead. Based on our observation (detail in Sec. 3.3), we find that these functions contain redundancies. Therefore, we try to further optimize the number of compound travel functions to improve the time complexity of the insertion.

3.3 Optimize Time Complexity to $O(n)$ by using Only $O(n)$ Compound Travel Functions

Observation: Two compound travel functions for each vertex v_j **is sufficient.** As Fig. 7 shows, we use $opt_i[d_2]$ records the optimal i when $j = 2$, *i.e.*, find the optimal position to insert o_{r^+} when inserting d_{r^+} before d_2. Based on the order constrain, there are only two cases for i value: (1) Fig. 7a, when $opt_i[d_2] = opt_i[o_2]$, the optimal i value is same as when insertion the d_{r^+} before the previous vertex o_2; (2) Fig. 7b, when $opt_i[d_2] \neq opt_i[o_2]$, the optimal i is updated. Based on the FIFO property, the determining factor is which case causes more time delay at d_2, as more time delay at d_2 results in more time delay at the last vertex v_n. To determine the optimal i value recorded in $opt_i[d_2]$, the delay time at d_2 can be calculated using one compound travel function from o_2 to d_2 in Fig. 7a, or using Eq. (4) in Fig. 7b. Then the increased travel time of the route can be queried by another compound travel function from d_2 to the last vertex d_1. Therefore, for d_2, two compound travel functions are sufficient to find the optimal insertion position pair $(opt_i[d_2], 2)$ before it.



(a) $opt_i[d_2] = 1 = opt_i[o_2]$



(b) $opt_i[d_2] = 2 \neq opt_i[o_2] = 1$

Figure 7: When $j = 2$, two cases for optimal i value

Basic Idea. The optimal insertion position i can be searched in $O(1)$ time by enumerating only the possible values of j. This is achieved by performing a delay time query from v_{j-1} to v_j. The increased travel time caused by delivering the new request before v_j then can be calculated by performing a delay time query from v_j to the last vertex v_n. Thus the worker only needs to maintain at most two compound travel functions for each vertex v_j in the route. As a result of the optimization, both the time and space costs are reduced from $O(n^2)$ to $O(n)$.

3.3.1 *Computing optimal i in $O(1)$ Time When Sequentially Increase j.*

We try to only enumerate the positions j to insert the destination of the new request while finding the optimal i, instead of enumerating all possible (i, j) pairs. If destination d_{r^+} is inserted before v_j in route S_R, then the origin o_{r^+} must also be inserted before v_j.

Calculate $opt_i[v_j]$ as i by One Compound Travel Function Between v_{j-1} and v_j. We use $opt_i[v_j]$ to record the optimal value of i when enumerating the j position along S_R. Specifically, by inserting o_{r^+} at $opt_i[v_j]$, we can achieve the shortest arrival time at vertex v_j, resulting in a better route with a shorter increased travel time due to the first-in-first-out (FIFO) property. Additionally, we use $arr'_{o_{r^+}}[v_j]$ to denote the arrival time at vertex v_j if we insert o_{r^+} at the position calculated before in $opt_i[v_{j-1}] \in \{i : 0 < i < j\}$, it can be calculated by $delayquery(v_{j-1}, v_j, *)$. If the new potential j-th position does not satisfy either the deadline constraint or capacity constraint for inserting o_{r^+}, then we set $opt_i[v_j] = NIL$. Otherwise, we can calculate the value based on the following lemma:

Lemma 3. Given a potential vertex v_j for inserting d_{r^+} before it, we update $opt_i[v_j] = j$ if and only if (1) $num[v_{j-1}] \leq c_w - c_{r^+}$; (2) $pik(o_{r^+}) \leq e_{r^+}$ and $query(o_{r^+}, v_j, pik(o_{r^+})) \leq latest[v_j]$; (3) $query(o_{r^+}, v_j, pik(o_{r^+})) < arr'_{o_{r^+}}[v_j]$. Here, $arr'_{o_{r^+}}[v_j]$ can be calculated using the delay time query from v_{j-1} to v_j as $arr'_{o_{r^+}}[v_{j-1}] + delayquery(v_{j-1}, v_j, arr'_{o_{r^+}}[v_{j-1}])$.

Proof. Condition (1) checks the capacity constraint and ensures that there is enough capacity for the worker to pick up r^+ at v_j. Condition (2) ensures that the deadline constraints are satisfied if o_{r^+} is inserted before v_j. Condition (3) guarantees that inserting o_{r^+} at the new potential j-th position results in a shorter travel time compared to inserting it in positions $\{i : 0 < i < j\}$. This can be proven by contradiction. Let us assume that the optimal position for inserting the new request's origin o_{r^+} is recorded in $opt_i[v_{j-1}]$ and lies within the set $\{i : 0 < i < j\}$. We insert o_{r^+} at $opt_i[v_{j-1}]$, the arrival time at v_j can be calculated by delay time query from v_{j-1} to v_j. Specifically, it can be expressed as $arr'_{o_{r^+}}[v_j] = arr'_{o_{r^+}}[v_{j-1}] + delayquery(v_{j-1}, v_j, arr'_{o_{r^+}}[v_{j-1}])$. However, if condition (3) is satisfied, inserting o_{r^+} at the new potential position j results in an earlier arrival time at v_j. If $opt_i[v_{j-1}]$ is the best position to insert o_{r^+}, we would obtain a route with a shorter travel time compared to inserting o_{r^+} before v_j. However, according to the FIFO property, if we choose $opt_i[v_{j-1}]$, the route would arrive later at v_j and the final vertex v_n with a larger delay, which contradicts the assumption. Therefore, j is a better position to insert o_{r^+}, and we update $opt_i[v_j] = j$. □

3.3.2 *Checking Constrains in $O(1)$ by using Only $O(n)$ Compound Travel Functions.*

We also need to check both capacity and deadline constraints. The capacity constraint check is similar to Lemma 2. After computing the optimal i value in $opt_i[v_j]$, we first update the new arrival time at v_j, then check the deadline constraint for inserting the new destination at position j.

Update New Arrival Time At v_j by One Compound Travel Function between v_{j-1} to v_j. We first update the arrival time at v_j based on $opt_i[v_j]$ for further feasibility checking. If $opt_i[v_j] = j$, we need to invoke the shortest arrival time query $query$ to calculate the new arrival time at v_j. Otherwise, if $opt_i[v_j] = opt_i[v_{j-1}]$, it indicates the the optimal position to insert o_{r^+} is one of the positions in $\{i : 0 < i < j\}$. To update the arrival time at v_j, we can rely on the delay time query from v_{j-1} to v_j.

The new arrival time at v_j is updated based on the following rules in $O(1)$ time. If $opt_i[v_j] = j$, o_{r^+} is also inserted at j-th position. Otherwise, $opt_i[v_j] = opt_i[v_{j-1}]$, o_{r^+} is inserted at one position in $\{i : 0 < i < j\}$:

$$arr'_{o_{r^+}}[v_j] = \begin{cases} query(o_{r^+}, v_j, pik(o_{r^+})), & opt_i[v_j] = j \\ arr_{o_{r^+}}[v_j] + delayquery(v_{j-1}, v_j, arr'_{o_{r^+}}[v_{j-1}]), & \text{otherwise} \end{cases} \quad (8)$$

Checking Constraints for v_j. As for the new destination d_{r^+}, before considering its insertion into the route at the j-th position, we must first verify its feasibility. This verification requires taking into account the worker's capacity constraint, the time constraint of the new request, and the new arrival time at v_j.

We first calculate the delivery time based on the updated arrival time at v_{j-1} after picking up the new request at position $opt_i[v_{j-1}]$. The delivery time is calculated as:

$$del(d_{r^+}) = query(v_{j-1}, d_{r^+}, arr'_{o_{r^+}}[v_{j-1}]) \quad (9)$$

Then, we update the arrival time at v_j after delivering the new request, by using the deliver time of the new request:

$$arr'[v_j] = query(d_{r^+}, v_j, del(d_{r^+})) \quad (10)$$

Table 3: $ComTravel$ **of** S_R **in Algo. 3**

	1	2
o_1	$ComTravel(o_1, o_2, t)$	$ComTravel(o_1, d_1, t)$
o_2	$ComTravel(o_2, d_2, t)$	$ComTravel(o_2, d_1, t)$
d_2	$ComTravel(d_2, d_1, t)$	×
d_1	×	×

Then we can perform the constraints verification, utilize the following lemma:

Lemma 4. The j-th position is a feasible position for inserting d_{r^+} if and only if (1) $del(d_{r^+}) = query(v_{j-1}, d_{r^+}, arr'_{o_{r^+}}[v_{j-1}]) \leq e_{r^+}$; (2) $arr'[v_j] = query(d_{r^+}, v_j, del(d_{r^+})) \leq latest[v_j]$; (3) $num[v_{j-1}] \leq c_w - c_{r^+}$

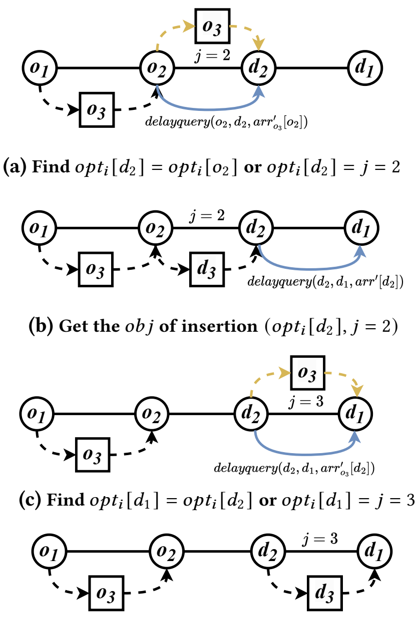
Proof. Condition (1) guarantees that inserting the new request's destination at the j-th position will not violate its deadline constraint. Following the insertion of d_{r^+} at the j-th position, condition (2) ensures that the new arrival time at v_j does not violate any deadline constraints for assigned requests delivered after v_j. Condition (3) checks the capacity constraint, ensuring that the worker's capacity is not exceeded. □

If Lemma 4 confirms that j is a feasible position to insert d_{r^+}, we follow Eq. (7) to calculate the increased travel time for the insertion position pair $(opt_i[v_j], j)$. The delay at the last vertex v_n is calculated by the delay time query which is supported by the compound travel function from v_j to the last vertex v_n. Therefore, we can calculate the increased travel time in $O(1)$ time.

Algorithm 3: Linear Time Insertion

Input: a worker w with a feasible route S_R and a request r^+
Output: new route S^* for w and min increased travel time obj^*

1 $S^* \leftarrow S_R, obj^* \leftarrow +\infty$;
2 Initialize $latest$, num by *Eq.* (5) - *Eq.* (6) ;
3 Compound travel functions $ComTravel(j, j+1, \cdot), ComTravel(j, n, \cdot), \forall j, 0 < j < n$ by Definition 5 ;
4 **for** $j \leftarrow 1$ *to* $n+1$ **do**
5   **if** *Lemma 3 (1) or (2) violated* **then** continue;
6   **if** *Lemma 3 (3)* **then** $opt_i[v_j] \leftarrow j$;
7   Update $arr'_{o_{r^+}}[v_j]$ after inserting o_{r^+} based on Eq. (8) ;
8   **if** *Lemma 4 (1)(2)(3)* **then**
9   Calculate $del(d_{r^+})$ after inserting o_{r^+} based on Eq. (9) ;
10   Calculate $arr'[v_j]$ of insertion $(opt_i[v_j], j)$ based on Eq. (10) ;
11         $obj \leftarrow$ *Eq.* (7) ;
12   **if** $obj < obj^*$ **then**
13             $obj^* \leftarrow obj, i^* \leftarrow opt_i[v_j], j^* \leftarrow j$;
14 **if** $obj^* < +\infty$ **then**
15     $S^* \leftarrow$ insert o_r at i^*-th and d_r at j^*-th in S_R ;
16   Get new arrival time of S^* by *Eq.* (4) ;
17 **return** S^*, obj^*;



(a) Find $opt_i[d_2] = opt_i[o_2]$ **or** $opt_i[d_2] = j = 2$

(b) Get the obj **of insertion** $(opt_i[d_2], j = 2)$

(c) Find $opt_i[d_1] = opt_i[d_2]$ **or** $opt_i[d_1] = j = 3$

(d) Get the obj **of insertion** $(opt_i[d_1], j = 3)$

Figure 8: Enumerating j **from 2 to 3**

3.3.3 *Linear Time Algorithm.*

Main Idea. We enumerate each potential j-th position in the worker's current route S_R. In each iteration, we use Lemma 3 to determine in $O(1)$ time whether inserting the new origin o_{r^+} at j is a better choice than other positions before v_j. The optimal position to insert o_{r^+} is then computed at $opt_i[v_j]$. Next, we check the feasibility of inserting the new destination d_{r^+} at position j based on Lemma 4 in $O(1)$ time. If it is feasible, we generate a new route by inserting o_{r^+} at $opt_i[v_j]$ and d_{r^+} at position j. The new arrival time at v_j after delivering the new request is calculated using Eq. (10) in $O(1)$ time. The increased travel time is calculated based on Eq. (7) in $O(1)$ time. All these calculations and feasibility checks have a time cost of $O(1)$.

Complexity Analysis. Line 3 of Algo. 3 computes the compound travel functions of route S_R. Line 4 takes $O(n)$ time to enumerate the potential position to insert d_{r^+}. We then check the constraints for inserting o_{r^+} at the new potential j-th position in line 5. If constraints are violated, the optimal position to insert o_{r^+} lies among the positions $\{i : 0 < i < j\}$. Otherwise, the j will be the best position in line 6. After o_{r^+} is inserted at $opt_i[v_j]$, the new arrival time at v_j is calculated in line 7. The feasibility of inserting d_{r^+} at j is checked, and the new arrival time at v_j is calculated in lines 8-10, each line takes $O(1)$ time. After inserting o_{r^+} at $opt_i[v_j]$ and d_{r^+} at j, the increased travel time of the newly generated route is calculated from lines 11-13. Therefore, the total time cost of Algo. 3 is $O(n)$. For the space cost, a worker maintains $O(n)$ compound travel functions, because each vertex v_j in the route requires at most 2 compound travel functions to be initialized.

Example 4. *Back to the settings in Fig. 1. For the current feasible route* $S_R = \langle o_1, o_2, d_2, d_1 \rangle$*, the compound travel functions along* S_R *are shown in Table 3. The algorithm computes (2+2+1) compound travel* *functions. For each vertex in S_R, at most two compound travel functions are maintained: one for the previous vertex and the other for the last vertex in the route. For the sake of presentation, we assume that insertion $(1, 3)$ is the optimal location pair to serve r_3. When $j = 1$, we determine that r_3 can only be picked up and delivered before o_2 thus $opt_i[o_2] = j = 1$. The detailed procedure of enumerating j from 2 to 3 is shown in Fig. 8. When $j = 2$, as shown in Fig. 8a-Fig. 8b, we first determine if inserting o_3 before d_2 ($opt_i[d_2] = 2$, marked in yellow) is a better choice than inserting o_3 before o_2 ($opt_i[d_2] = opt_i[o_2]$), which is shown in Fig. 8a. To compare the new arrival times at vertex d_2, we consider two cases. If we insert o_3 at position 1, the new arrival time is computed using the delay time query from o_2 to its next vertex d_2: $arr'_{o_3}[d_2] + delayquery(o_2, d_2, arr'_{o_3}[o_2])$. Here, $arr'_{o_3}[o_2]$ is the arrival time at o_2 if o_3 is inserted before it. Alternatively, if we insert o_3 at position 2, the new arrival time at d_2 can be computed as $query(o_3, d_2, pik(o_3))$. Following the assumption, by comparing the two arrival times, we obtain $query(o_3, d_2, pik(o_3)) > arr'_{o_3}[d_2] + delayquery(o_2, d_2, arr'_{o_3}[o_2])$. Therefore, we conclude that inserting o_3 at position 1 is optimal for all positions before d_2, thus we have $opt_i[d_2] = opt_i[o_2] = 1$. Next, as shown in Fig. 8b, we calculate the increased travel time caused by insertion pair ($opt_i[d_2] = 1, j = 2$) for $j = 2$. The computation is performed using the delay time query from d_2 to the last vertex d_1: $arr[d_1] + delayquery(d_2, d_1, arr'[d_2])$, where $arr'[d_2]$ is the arrival time at d_2 after delivering the new request before it. When $j = 3$, as shown in Fig. 8c-Fig. 8d, we first evaluate $opt_i[d_1]$ for inserting o_3 following a similar procedure as before. In the evaluation, we use the delay time query $delayquery(d_2, d_1, arr'_{o_3}[d_2])$, where $arr'_{o_3}[d_2]$ is the arrival time at d_2 if o_3 is inserted at $opt_i[d_2]$. We then obtain the increased travel time caused by insertion pair ($opt_i[d_1] = 1, j = 3$), and compare it with the value of pair ($opt_i[d_2] = 1, j = 2$) in Fig. 8b. The result shows that this is the best insertion pair to serve the new request r_3.*

Correctness. The algorithm can find the optimal position pair $(opt_i[v_j], j)$ with the minimum increased travel time to deliver the assigned new request r^+. The order constraint can be verified by Lemma 3. The feasibility of $(opt_i[v_j], j)$ for insertion is ensured by the correctness of Lemma 3 - Lemma 4, while *Eq.* (4) - *Eq.* (7) guarantee the minimum increased travel time.

4 EXPERIMENT STUDY

In this section, we first present the experimental setup for our study, then we demonstrate our experimental results.

4.1 Experimental Setup

Datasets. The proposed methods are evaluated on three datasets from different applications, with road networks downloaded from OpenStreetMap [5]. The number of edges varies from 80,000 to over 900,000, with a time domain of 86,400 seconds (a whole day). Time-dependent weight functions have 518,282, 138,083, and 1,411,569 interpolation points, respectively. The TDSP query function in [36] is used as the shortest arrival time query *query*. Statistical characteristics of the datasets are presented in Table 4.

The first application considered is **Ridesharing**, using two public datasets from Chengdu and Haikou City, China denoted as ***Chengdu*** and ***Haikou***, respectively. The dataset consists of more than 200,000 requests during daytime hours(i.e., 8:00 am to 18:00 pm), with origins and destinations mapped to the nearest vertex of the road network. The time period $e_r - t_r$ from each request release time to deadline ranges from 10 to 30 minutes, with worker capacities varying from 3 to 20, which is consistent with the setting in existing works [33, 38] for the capacity of workers in ridesharing service. Requests released at different times of the day are used to test the methods with varying numbers of requests, ranging from 2 to 10 hours (i.e., 8:00 am-10:00 am (2h), 8:00 am-12:00 pm (4h), $\cdots$ 8:00 am to 18:00 pm (10h)).

Table 4: Statistics of datasets.

Dataset		#(Vertices)	#(Edges)	#(Interpolation)
Ridesharing	Chengdu	423,434	913,718	1,411,569
	Haikou	41,542	89,206	138,083
Logistics	Cainiao	9,936	23,872	518,282

Table 5: Parameter settings.

Parameters	Settings
Capacity c_w	Ridesharing: 3, **5**, 10, 15, 20
	Logistics: 80, **100**, 120, 140, 160
Requests release time duration:	Ridesharing: 2h, **4h**, 6h, 8h, 10h
	Logistics: 4h, **6h**, 8h, 10h, 12h
Time period: $e_r - t_r$ (minute)	Ridesharing: 10, **15**, 20, 25, 30
	Logistics: original ddl information
Scalability: # of workers	Ridesharing : 100, **200**, 300, 400, 500
	Logistics: 2k, **4k**, 6k, 8k, 10k

Table 6: Proposed Algorithms.

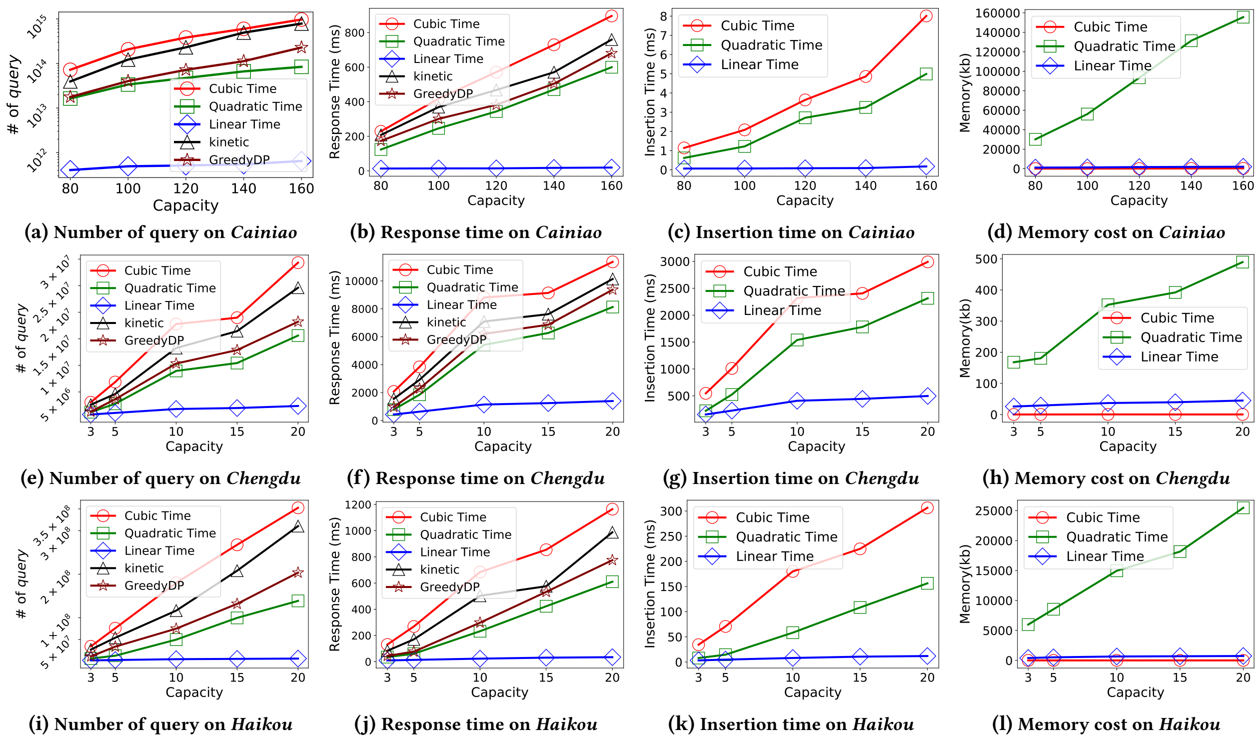
Algorithms	Time Complexity	Space Complexity
Cubic Time Algorithm	$O(n^3)$	$O(n)$
Quadratic Time Algorithm	$O(n^2)$	$O(n^2)$
Linear Time Algorithm	$O(n)$	$O(n)$

The second application is **Logistics**, using a dataset collected in Shanghai by the Cainiao logistics platform [1]. The dataset contains parcel origins, destinations, and deadlines, and is preprocessed by downloading the road network map of Shanghai City. The original deadlines provided in the dataset are utilized, with parameter settings summarized in Table 5.

Compared Algorithms. We compare the following algorithms in this section. Table 6 summaries the algorithms we propose.

- *Cubic Time Algorithm (Algo. 1 of this paper)* [38]. The baseline method implements the insertion operator over time-dependent road networks.
- *kinetic* [17]. A kinetic tree maintains all possible routes to serve the request.
- *GreedyDP* [34]. A dynamic programming based algorithm to plan routes of requests.
- *Quadratic Time Algorithm (Algo. 2 of this paper).* It generates new potential routes by enumerating all possible (i, j) pairs.
- *Linear Time Algorithm (Algo. 3 of this paper).* It enumerates j and find the optimal i in $O(1)$ time.

Metrics. To evaluate time-dependent insertion performance, we vary the following values (1) worker's capacity c_w; (2) time period $e_r - t_r$ for each request; (3) requests release time duration; (4) the number of workers to study the scalability. We choose the metrics requiring evaluation: (1) the number of invoking shortest arrival time queries *query*; (2) insertion time; (3) response time; (4) memory



(a) Number of query on *Cainiao* **(b) Response time on *Cainiao*** **(c) Insertion time on *Cainiao*** **(d) Memory cost on *Cainiao***

(e) Number of query on *Chengdu* **(f) Response time on *Chengdu*** **(g) Insertion time on *Chengdu*** **(h) Memory cost on *Chengdu***

(i) Number of query on *Haikou* **(j) Response time on *Haikou*** **(k) Insertion time on *Haikou*** **(l) Memory cost on *Haikou***

Figure 9: Results of varying capacity of the worker

cost. The bottleneck of time-dependent insertion is the shortest arrival time query over the time-dependent road network, and we aim to improve efficiency by avoiding invoking a large number of *query*. In terms of the insertion time, for *Cubic Time Algorithm* and *Quadratic Time Algorithm*, it indicates the average time required to check the feasibility and calculate the increased travel time for each possible (i, j) pair. For *Linear Time Algorithm*, it represents the average time for each j value to check the feasibility, then find the optimal i, then calculate the increased travel time. Response time and memory cost are both commonly used metrics in many shared mobility applications [24][17][34]. Response time is the average time required to assign each request. Memory cost for *Cubic Time Algorithm* is for storing the worker's route, while for *Quadratic Time Algorithm* and *Linear Time Algorithm*, it's dominated by maintained compound travel functions and auxiliary arrays.

The *kinetic* and *GreedyDP* methods are specifically designed for static road networks. To adapt them for use in time-dependent road networks, we modify them by using *query* to check the feasibility and calculate the arrival time of each vertex in a new possible route, and the number of invoking *query* is reported. Ref. [17, 34] use the response time as the metric of these methods, and since *kinetic* is not a method based on the insertion operator, we report the response time to show the efficiency. Memory cost is omitted, as these methods use only a few auxiliary arrays.

Implementation. 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 algorithms are implemented in GNU C++. Each experiment is repeated 10 times and the average results are reported.

4.2 Experimental Results

Impact of Capacity of Workers. Fig. 9 shows the impact of worker's capacity on three datasets. Compared to the baseline method, *Linear Time Algorithm* invokes the shortest arrival time queries much less (75.24% - 97.72% reduction), demonstrating the effectiveness of the compound travel function. *Linear Time Algorithm* also has the fastest insertion time (up to 44.5 times faster on ***Cainiao*** than *Cubic Time Algorithm*) and reduces memory cost by 90.8% - 98.7% compared to *Quadratic Time Algorithm*. As capacity increases, time and memory costs increase, but *Linear Time Algorithm* consistently has the lowest time cost and much lower memory cost (no more than 45 KB on ***Chengdu***) than *Quadratic Time Algorithm*. Note that the insertion time and response time of *Cubic Time Algorithm*, and the memory cost of *Quadratic Time Algorithm* vary significantly due to time and space complexity.

Impact of Time $e_r - t_r$. Fig. 10 shows the results of varying the deadline of requests on ***Haikou*** and ***Chengdu***. The values of $e_r - t_r$ are the scale values of the x-axis. *Linear Time Algorithm* outperforms others in terms of efficiency, which is up to 16.96 times faster in response time. As $e_r - t_r$ increases, requests have larger deadlines, more requests can be inserted into the original route, and the time cost of *Cubic Time Algorithm* and space cost of *Quadratic Time Algorithm* increase significantly. Compared to the method with $O(n)$ space cost, *Linear Time Algorithm* consumes slightly higher memory (no more than 1 MB, which is acceptable).

Impact of Requests Release Time Duration. Fig. 11 shows the impact of release time duration on ***Cainiao*** and ***Chengdu***. *Linear Time Algorithm* remains the fastest method with significantly fewer *query* invocations (570.60 and 6.21 times fewer) and faster

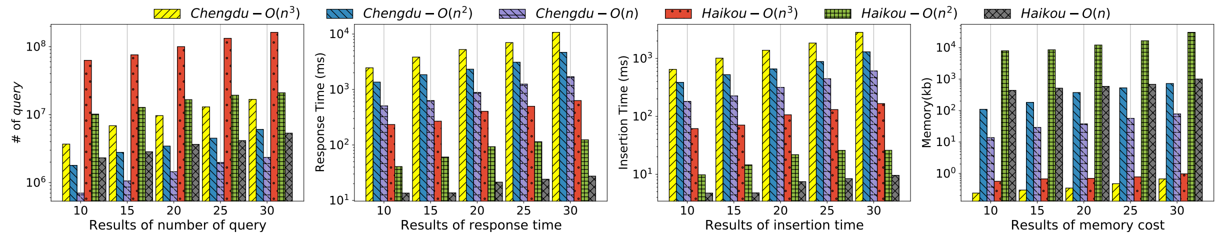
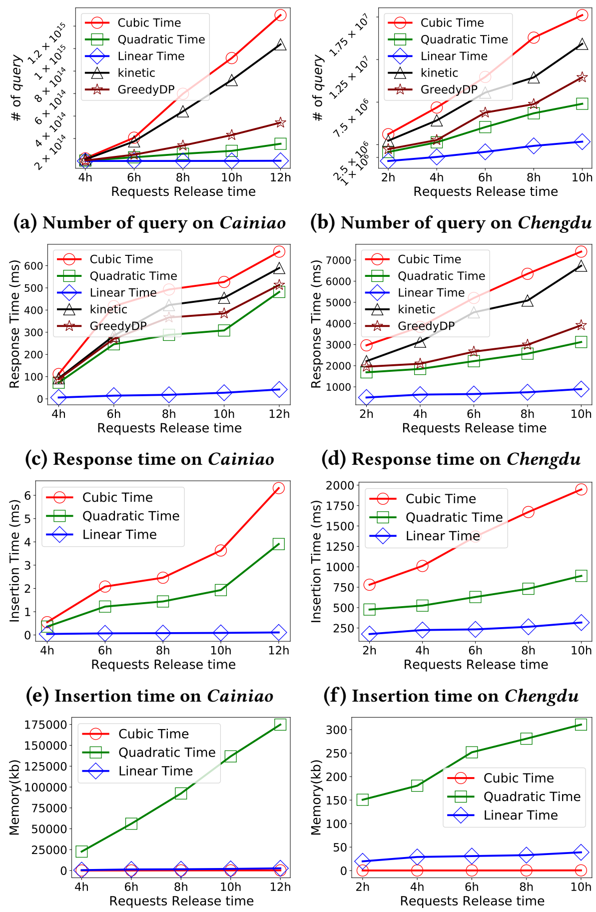


Figure 10: Results of varying $e_r - t_r$



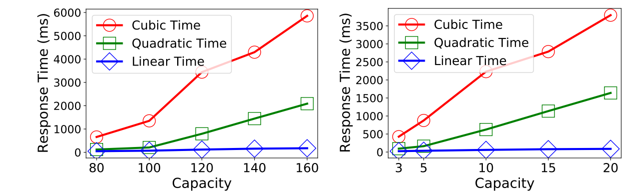
(a) Number of query on *Cainiao* **(b) Number of query on *Chengdu***

(c) Response time on *Cainiao* **(d) Response time on *Chengdu***

(e) Insertion time on *Cainiao* **(f) Insertion time on *Chengdu***

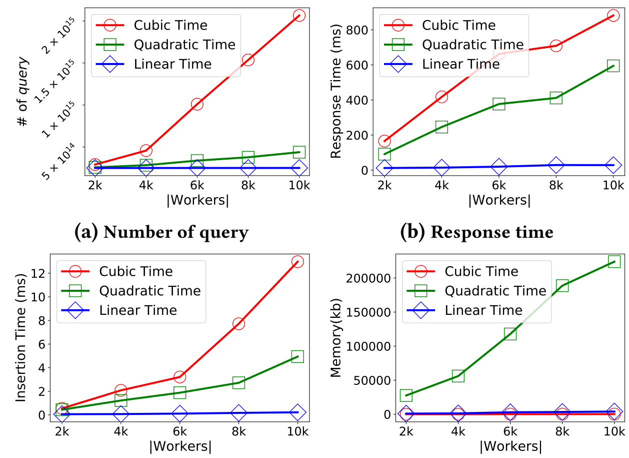
(g) Memory cost on *Cainiao* **(h) Memory cost on *Chengdu***

Figure 11: Results of varying requests release time duration



(a) Response time on *Cainiao* **(b) Response time on *Haikou***

Figure 12: Results when 10% of road condition changes in the road networks.



(a) Number of query **(b) Response time**

(c) Insertion time **(d) Memory cost**

Figure 13: Results of scalability test on *Cainiao*

response time (15.78 and 6.09 times faster) than the baseline on the two datasets. As release time duration increases, the number of requests increases, and time cost increases for all algorithms. *Quadratic Time Algorithm* still has the highest memory cost, while the memory cost of *Linear Time Algorithm* slightly increases.

Impact of Changes in Road Conditions. We have tested the algorithms for planning routes for shared mobility services, while following the same procedures in Ref. [36] to randomly change 10% road conditions of the road networks. The response time results for ***Cainiao*** and ***Haikou*** are reported in Fig. 12, and we have observed that the *Linear Time Algorithm* remains the most efficient.

Scalability Test. This experiment tests the scalability of time-dependent insertion, with results of ***Cainiao*** shown in Fig. 13. The number of *query* invocations grows exponentially with increasing workers, causing slower response time for baselines. This shows that the existing insertion operator can be a bottleneck in time-dependent road networks. For *Quadratic Time Algorithm*, memory cost becomes notably higher when the number of workers is larger. Except for memory cost, *Linear Time Algorithm* outperforms the other two methods in all metrics, enabling it to assign requests in real-time. *Linear Time Algorithm* takes no more than 4 MB memory cost. These results demonstrate the suitability of our proposed method for real large-scale road networks.

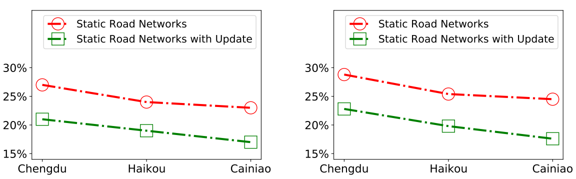
Computation Cost of $ComTravel()$. Table 7 presents the average computation time of the compound travel function for each new route, it is included in the average *Responsetime*. Results are reported for both default and maximum capacity settings, where more capacity leads to longer routes and more complex functions. The results demonstrate the efficiency and effectiveness of the proposed compound travel function.

Case Study: Benefits of Time-Dependent Networks. Time-dependent road networks exhibit predictable characteristics, enabling foresight and prediction of future road network changes

Table 7: Cost of $ComTravel()$ for ***Linear Time Algo.***

	Haikou		***Chengdu***		***Cainiao***	
Worker's capacity c_w	5	20	5	20	100	160
Computation time (*ms*)	6.50	11.79	59.43	179.46	9.72	14.42
Occupy ration to *Response time*	28%	34%	9%	12%	22%	26%

during route planning [19, 42], thus offering potential benefits in future planning. However, existing methods for insertion operations are not always optimal in realistic dynamic settings. To demonstrate this, we conduct experiments in the following road networks: **(1) Static Road Networks**, where we compute the average travel time of each edge over the whole day as the static constant travel time of the edge in the static scenario; **(2) Static Road Networks with Update**, where route updates are conducted when there is an edge travel time change. We use the state-of-the-art method DCH-WInc [27] to compute and maintain the travel time of the route.



(a) Requests release time $4h$ **(b) Requests release time** $10h$

Figure 14: Requests are delayed in comparison to time-dependent road networks.

Fig. 14 shows the results of requests delay. We observe that static road networks can cause more than 20% of requests to not reach their destinations before deadlines, resulting in a reduction in the number of served requests compared to time-dependent road networks. Notably, the dynamic update road networks can also cause more than 15% delayed requests, underscoring the importance of the data summary in accurately predicting future changes during route planning and offering potential benefits.

Summary of Experiments. After conducting extensive experiments, we summarize the results as follows:

- It is impractical to extend the existing insertion operator straightforwardly to the time-dependent road networks. None of the baselines are more efficient than the $O(n^2)$ algorithm. The baseline method takes more than 11,300 milliseconds to respond to one request on ***Chengdu*** and more than 890 milliseconds to respond to one request on ***Cainiao***. Therefore, it is impractical to handle requests from real-world shared mobility services.
- Our optimized algorithms, *Quadratic Time Algorithm* and *Linear Time Algorithm*, are 2.16 - 25.67 times faster for *Ride−sharing* service and 6.39 - 44.5 times faster for *Logistics* service compared to the baseline, due to their improvement in time complexity.
- *Linear Time Algorithm* reduces memory cost by up to 97.1% compared with *Quadratic Time Algorithm*. The memory consumed by *Linear Time Algorithm* is only slightly larger than *Cubic Time Algorithm* for both *Ridesharing* and *Logistics* services, *e.g.*, no more than 77 KB on ***Chengdu***.

5 RELATED WORK

Route processing over time-dependent road networks. The time-dependent road network has attracted many research efforts in spatial databases. To model dynamic traffic in reality, piecewise linear functions are widely adopted to fit the time-dependent edge weight functions [36] [8] [37]. Novel indexes are introduced in [14, 36] to answer route planning queries. Ref. [14] proposes a dual-level path index and utilizes a filter-and-refine strategy to enhance efficiency. Ref. [36] splits the road network into hierarchical partitions and constructed a balanced tree index, then it computes the travel cost functions of border vertexes in partitions to answer queries.

Route planning over time-dependent road networks is another critical problem. Ref. [8] proposes an online route planning problem and develops a request-inserted algorithm to reduce the competitive ratio. However, it assumes that all passengers have the same destination. In [13], the last mile delivery problem is extended to time-dependent road networks, where a courier takes multiple parcels starting from the same warehouse, and each parcel can be delivered to alternative locations depending on the time. An insertion based method is proposed to heuristically insert the delivery location into the courier's path to maximize the number of delivered parcels. The existing works make the assumption that all passengers have the same destination or parcels have the same origin. In contrast, our time-dependent insertion problem is more challenging as it requires a feasible route, including both the origin and destination of each request. As shown in example 1, a new feasible route including both the origin o_3 and destination d_3 is planned to serve the new request r_3. Furthermore, [8] and [13] omit to check the feasibility of the worker. Therefore, existing algorithms cannot be applied to our problem.

Route planning for shared mobility. The shared mobility service has also been studied in many domains varying from Database to AI. Different studies focus on different objectives, [21, 24] focus on maximizing the number of requests served, while [10, 16, 17, 29, 38] focus on minimizing the total travel time. Additionally, from the platform's perspective, the total revenue is also an objective that needs to be taken into consideration, as shown in [44, 45].

The insertion operator is an efficient method for planning new routes by inserting a new request into a worker's current route, as proposed in [31]. Recent studies have demonstrated its effectiveness and efficiency in large-scale static road networks [34, 38, 39]. Ref. [38, 39] extensively studied the insertion operator and developed a dynamic programming-based partition framework that reduces time complexity from $O(n^3)$ to $O(n^2)$, and further reduces it to $O(n)$ using a fenwick tree index. In [34], a unified route planning problem for shared mobility is defined, and a novel two-phased solution based on dynamic programming insertion is proposed to solve it approximately. However, these solutions do the calculations following *additivity property* of the static road networks, none of them consider the time-varying travel cost characteristics of road networks in real-world applications.

6 CONCLUSION

In this paper, we first study a real-time insertion operator for shared mobility on time-dependent road networks. A novel data summary model is designed to support the delay time query along the worker's route. The model is built by computing the compound travel functions between vertices of the route. By leveraging the query, we can reduce the time complexity of the existing insertion operator extended to the dynamic scenario from $O(n^3)$ to $O(n^2)$. Furthermore, we achieve an insertion operator with linear time and space cost by exploiting the FIFO property of a time-dependent road network. Specifically, we prove that given a position to insert the destination, we can find the optimal position to insert the origin of the new request in $O(1)$ time. Extensive experiments on datasets from different real-world applications demonstrate the efficiency and scalability of our time-dependent insertion operator. In particular, our proposed insertion operator can be up to 44.5 times faster than the baseline method under different settings on three real-world datasets.

REFERENCES

[1] 2023. Cainiao. https://global.cainiao.com/
[2] 2023. DiDi Chuxing. https://www.didiglobal.com/
[3] 2023. Google Map. https://www.google.com/maps/
[4] 2023. Meituan. https://www.meituan.com
[5] 2023. Openstreetmap. http://www.openstreetmap.com/
[6] 2023. What is Shared Mobility? https://sharedusemobilitycenter.org/what-is-shared-mobility/
[7] Mohammad Asghari and Cyrus Shahabi. 2017. An On-Line Truthful and Individually Rational Pricing Mechanism for Ride-Sharing *(SIGSPATIAL '17)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/3139958.3139991
[8] Di Chen, Ye Yuan, Wenjin Du, Yurong Cheng, and Guoren Wang. 2021. Online Route Planning over Time-Dependent Road Networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 325–335. https://doi.org/10.1109/ICDE51399.2021.00035
[9] Di Chen, Ye Yuan, Wenjin Du, Yurong Cheng, and Guoren Wang. 2021. Online Route Planning over Time-Dependent Road Networks. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 325–335. https://doi.org/10.1109/ICDE51399.2021.00035
[10] Lu Chen, Qilu Zhong, Xiaokui Xiao, Yunjun Gao, Pengfei Jin, and Christian S. Jensen. 2018. Price-and-Time-Aware Dynamic Ridesharing. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1061–1072. https://doi.org/10.1109/ICDE.2018.00099
[11] Peng Cheng, Hao Xin, and Lei Chen. 2017. Utility-Aware Ridesharing on Road Networks. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1197–1210. https://doi.org/10.1145/3035918.3064008
[12] Blerim Cici, Athina Markopoulou, and Nikolaos Laoutaris. 2015. Designing an On-Line Ride-Sharing System. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Seattle, Washington) *(SIGSPATIAL '15)*. Association for Computing Machinery, New York, NY, USA, Article 60, 4 pages. https://doi.org/10.1145/2820783.2820850
[13] Camila F. Costa and Mario A. Nascimento. 2021. Last Mile Delivery Considering Time-Dependent Locations. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems* (Beijing, China) *(SIGSPATIAL '21)*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/3474717.3483919
[14] Tianlun Dai, Bohan Li, Ziqiang Yu, Xiangrong Tong, Meng Chen, and Gang Chen. 2021. PARP: A Parallel Traffic Condition Driven Route Planning Model on Dynamic Road Networks. *ACM Trans. Intell. Syst. Technol.* 12, 6, Article 73 (dec 2021), 24 pages. https://doi.org/10.1145/3459099
[15] Esteban Feuerstein and Leen Stougie. 2001. On-Line Single-Server Dial-a-Ride Problems. *Theor. Comput. Sci.* 268, 1 (oct 2001), 91–105. https://doi.org/10.1016/S0304-3975(00)00261-9
[16] Wesam Mohamed Herbawi and Michael Weber. 2012. 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* (Philadelphia, Pennsylvania, USA) *(GECCO '12)*. Association for Computing Machinery, New York, NY, USA, 385–392. https://doi.org/10.1145/2330163.2330219
[17] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. 2014. Large Scale Real-Time Ridesharing with Service Guarantee on Road Networks. *Proc. VLDB Endow.* 7, 14 (oct 2014), 2017–2028. https://doi.org/10.14778/2733085.2733106
[18] Lauri Häme. 2011. An adaptive insertion algorithm for the single-vehicle dial-a-ride problem with narrow time windows. *European Journal of Operational Research* 209, 1 (2011), 11–22. https://doi.org/10.1016/j.ejor.2010.08.021
[19] Ke Li, Xuan Rao, Xiaobing Pang, Lisi Chen, and Siqi Fan. 2021. Route Search and Planning: A Survey. *Big Data Res.* 26 (2021), 100246. https://doi.org/10.1016/J.BDR.2021.100246
[20] Yafei Li, Ji Wan, Rui Chen, Jianliang Xu, Xiaoyi Fu, Hongyan Gu, Pei Lv, and Mingliang Xu. 2021. Top-k Vehicle Matching in Social Ridesharing: A Price-Aware Approach. *IEEE Transactions on Knowledge and Data Engineering* 33, 3 (2021), 1251–1263. https://doi.org/10.1109/TKDE.2019.2937031
[21] Zhidan Liu, Zengyang Gong, Jiangzhou Li, and Kaishun Wu. 2020. Mobility-Aware Dynamic Taxi Ridesharing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 961–972. https://doi.org/10.1109/ICDE48307.2020.00088
[22] Hui Luo, Zhifeng Bao, Farhana M. Choudhury, and J. Shane Culpepper. 2021. Dynamic Ridesharing in Peak Travel Periods. *IEEE Transactions on Knowledge and Data Engineering* 33, 7 (2021), 2888–2902. https://doi.org/10.1109/TKDE.2019.2961341
[23] Wenjun Lyu, Kexin Zhang, Baoshen Guo, Zhiqing Hong, Guang Yang, Guang Wang, Yu Yang, Yunhuai Liu, and Desheng Zhang. 2022. Towards Fair Workload Assessment via Homogeneous Order Grouping in Last-Mile Delivery. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management* (Atlanta, GA, USA) *(CIKM '22)*. Association for Computing Machinery, New York, NY, USA, 3361–3370. https://doi.org/10.1145/3511808.3557132
[24] Shuo Ma, Yu Zheng, and Ouri Wolfson. 2013. T-share: A large-scale dynamic taxi ridesharing service. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 410–421. https://doi.org/10.1109/ICDE.2013.6544843
[25] Shuo Ma, Yu Zheng, and Ouri Wolfson. 2015. Real-Time City-Scale Taxi Ridesharing. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1782–1795. https://doi.org/10.1109/TKDE.2014.2334313
[26] Masayo Ota, Huy Vo, Cláudio Silva, and Juliana Freire. 2017. STaRS: Simulating Taxi Ride Sharing at Scale. *IEEE Transactions on Big Data* 3, 3 (2017), 349–361. https://doi.org/10.1109/TBDATA.2016.2627223
[27] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient Shortest Path Index Maintenance on Dynamic Road Networks with Theoretical Guarantees. *Proc. VLDB Endow.* 13, 5 (2020), 602–615. https://doi.org/10.14778/3377369.3377371
[28] Zhiwei Tony Qin, Hongtu Zhu, and Jieping Ye. 2021. Reinforcement Learning for Ridesharing: A Survey. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. 2447–2454. https://doi.org/10.1109/ITSC48978.2021.9564924
[29] Douglas O. Santos and Eduardo C. Xavier. 2013. Dynamic Taxi and Ridesharing: A Framework and Heuristics for the Optimization Problem. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (Beijing, China) *(IJCAI '13)*. AAAI Press, 2885–2891.
[30] Susan Shaheen, Adam Cohen, Ismail Zohdy, and Beaudry Kock. 2016. Shared Mobility: Current Practices and Guiding Principles Brief. (2016).
[31] Qian Tao, Yuxiang Zeng, Zimu Zhou, Yongxin Tong, Lei Chen, and Ke Xu. 2018. Multi-Worker-Aware Task Planning in Real-Time Spatial Crowdsourcing. In *Database Systems for Advanced Applications*, Jian Pei, Yannis Manolopoulos, Shazia Sadiq, and Jianxin Li (Eds.). Springer International Publishing, Cham, 301–317.
[32] Raja Subramaniam Thangaraj, Koyel Mukherjee, Gurulingesh Raravi, Asmita Metrewar, Narendra Annamaneni, and Koushik Chattopadhyay. 2017. Xhare-a-Ride: A Search Optimized Dynamic Ride Sharing System with Approximation Guarantee. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 1117–1128. https://doi.org/10.1109/ICDE.2017.156
[33] Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, and Ke Xu. 2022. Unified Route Planning for Shared Mobility: An Insertion-Based Framework. *ACM Trans. Database Syst.* 47, 1, Article 2 (may 2022), 48 pages. https://doi.org/10.1145/3488723
[34] Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, Jieping Ye, and Ke Xu. 2018. A Unified Approach to Route Planning for Shared Mobility. *Proc. VLDB Endow.* 11, 11 (jul 2018), 1633–1646. https://doi.org/10.14778/3236187.3236211
[35] Jiachuan Wang, Peng Cheng, Libin Zheng, Lei Chen, and Wenjie Zhang. 2022. Online Ridesharing with Meeting Points. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3963–3975.
[36] Yong Wang, Guoliang Li, and Nan Tang. 2019. Querying Shortest Paths on Time Dependent Road Networks. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1249–1261. https://doi.org/10.14778/3342263.3342265
[37] Yishu Wang, Ye Yuan, Hao Wang, Xiangmin Zhou, Congcong Mu, and Guoren Wang. 2021. Constrained Route Planning over Large Multi-Modal Time-Dependent Networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 313–324.

https://doi.org/10.1109/ICDE51399.2021.00034
[38] Yi Xu, Yongxin Tong, Yexuan Shi, Qian Tao, Ke Xu, and Wei Li. 2019. An Efficient Insertion Operator in Dynamic Ridesharing Services. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1022–1033. https://doi.org/10.1109/ICDE.2019.00095
[39] Yi Xu, Yongxin Tong, Yexuan Shi, Qian Tao, Ke Xu, and Wei Li. 2020. An Efficient Insertion Operator in Dynamic Ridesharing Services. *IEEE Transactions on Knowledge and Data Engineering* (2020), 1–1. https://doi.org/10.1109/TKDE.2020.3027200
[40] Jieping Ye. 2018. Big Data at Didi Chuxing. In *The 41st International ACM SIGIR Conference on Research and Development in Information Retrieval* (Ann Arbor, MI, USA) *(SIGIR '18)*. Association for Computing Machinery, New York, NY, USA, 1341. https://doi.org/10.1145/3209978.3210213
[41] Jieping Ye. 2019. Transportation: A Data Driven Approach. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 3183. https://doi.org/10.1145/3292500.3340406
[42] Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2021. An Experimental Evaluation and Guideline for Path Finding in Weighted Dynamic Network. *Proc. VLDB Endow.* 14, 11 (2021), 2127–2140. https://doi.org/10.14778/3476249.3476267
[43] Bolong Zheng, Chenze Huang, Christian S. Jensen, Lu Chen, Nguyen Quoc Viet Hung, Guanfeng Liu, Guohui Li, and Kai Zheng. 2020. Online Trichromatic Pickup and Delivery Scheduling in Spatial Crowdsourcing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 973–984. https://doi.org/10.1109/ICDE48307.2020.00089
[44] Libin Zheng, Lei Chen, and Jieping Ye. 2018. Order Dispatch in Price-Aware Ridesharing. *Proc. VLDB Endow.* 11, 8 (apr 2018), 853–865. https://doi.org/10.14778/3204028.3204030
[45] Libin Zheng, Peng Cheng, and Lei Chen. 2019. Auction-Based Order Dispatch and Pricing in Ridesharing. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1034–1045. https://doi.org/10.1109/ICDE.2019.00096
[46] Guanzhou Zhu, Dong Zhao, Yizong Wang, Haotian Wang, Desheng Zhang, and Huadong Ma. 2023. COME: Learning to Coordinate Crowdsourcing and Regular Couriers for Offline Delivery During Online Mega Sale Days. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 3126–3139. https://doi.org/10.1109/ICDE55515.2023.00240