2019 IEEE 35th International Conference on Data Engineering (ICDE)

An Efficient Insertion Operator in Dynamic Ridesharing Services

Yi Xu [†], Yongxin Tong [†], Yexuan Shi [†], Qian Tao [†], Ke Xu [†], Wei Li [‡]
State Key Laboratory of Software Development Environment and
Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, China
[†] {xuy, yxtong, skyxuan, qiantao, kexu}@buaa.edu.cn, [‡] liwei@nlsde.buaa.edu.cn

***Abstract*—Dynamic ridesharing refers to services that arrange one-time shared rides on short notice. It underpins various real-world intelligent transportation applications such as car-pooling, food delivery and last-mile logistics. A core operation in dynamic ridesharing is the "*insertion operator*". Given a worker and a feasible route which contains a sequence of origin-destination pairs from previous requests, the insertion operator inserts a new origin-destination pair from a newly arrived request into the current route such that certain objective is optimized. Common optimization objectives include minimizing the maximum flow time of all requests and minimizing the total travel time of the worker. Despite its frequent usage, the insertion operator has a time complexity of $O(n^3)$, where n is the number of all requests assigned to the worker. The cubic running time of insertion fundamentally limits the efficiency of urban-scale dynamic ridesharing based applications. In this paper, we propose a novel partition framework and a dynamic programming based insertion with a time complexity of $O(n^2)$. We further improve the time efficiency of the insertion operator to $O(n)$ harnessing efficient index structures, such as fenwick tree. Evaluations on two real-world large-scale datasets show that our methods can accelerate insertion by 1.5 to 998.1 times.**

I. Introduction

Dynamic ridesharing refers to services that arrange one-time shared rides on short notice. It underpins various real-world intelligent transportation applications such as car-pooling, food delivery and last-mile logistics [1]. For a set of workers and a sequence of dynamic requests, one primary function in dynamic ridesharing is to arrange for each worker a route to pick up and drop off requests. A worker can be a driver in car-pooling or a courier in food delivery and logistics, while a request can be one or multiple passengers or parcels accordingly. Dynamic ridesharing has been extensively studied in the database community [2], [3], [4], [5], [6], [7], [8]. It has been proved that there is no polynomial-time algorithm with a constant competitive ratio to solve the problem [8]. Hence many real-world ridesharing platforms, such as Didi Chuxing and Uber, rely on heuristic algorithms [2], [4], [5], [8].

Insertion, or an "*insertion operator*", is widely adopted in various heuristic solutions to dynamic ridesharing [9], [10], [11], [12], [2], [4], [5], [8] and is recognized as a core operator in these solutions [13], [14], [15], [16]. Given a worker and a feasible route which contains a sequence of origin-destination pairs from previous requests, insertion, *a.k.a.* an insertion operator, inserts a new origin-destination pair from a newly arrived request into the current route such that certain objective is optimized. The objective of a generic insertion operator is defined from the perspective of either the requests or the worker. From the requests' perspective, insertion needs to minimize the maximum waiting time/distance of all the requests. From the workers' perspective, insertion should minimize the total travel time/distance of the worker.

Despite its importance, the generic insertion operator remains an efficiency bottleneck for dynamic ridesharing algorithms. The insertion that optimizes from the requests' perspective has a time complexity of $O(n^3)$, where n is the number of all the requests for the worker. The cubic running time limits the efficiency of urban-scale dynamic ridesharing based applications. Although a linear-time insertion algorithm that optimizes the objective from the workers' perspective has been proposed [8], it cannot be adapted for the optimization objective from the requests' perspective as the linear-time insertion algorithm in [8] is derived from a special recursion relationship for the objective from the workers' perspective.

To break the efficiency bottleneck, we propose a partition-based framework and devise an $O(n^2)$-time insertion operator. In addition, we harness efficient index structures, such as the fenwick tree [17], and further reduce the time complexity of a generic insertion operator to linear time.

Our main contributions can be summarized as follows.

- We systematically study the generic insertion operator for dynamic ridesharing and propose a partition-based framework to reduce the time complexity of a generic insertion operator to $O(n^2)$.
- Based on the partition-based framework, we further improve the time efficiency of the insertion operator to $O(n)$ utilizing efficient index structures, such as fenwick tree.
- Experimental results show that our algorithms can speed up the insertion operator by 1.5 to 998.1 times on real-world urban-scale datasets.

In the rest of this paper, we formally introduce the insertion operator in Sec. II and review existing solutions in Sec. III. We propose a partition-based framework in Sec. IV and design a series of linear-time optimization techniques to reduce the time complexity of insertion in Sec. V. Finally we present the evaluations in Sec. VI and conclude in Sec. VII.

II. Problem Statement

This section presents the generic formulation of the insertion operator in ridesharing services.

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00095

Definition 1 (Worker)**.** *A worker is defined as $w =< o_w, c_w >$ with a current location of o_w and a capacity of c_w, where the capacity is the maximum number of passengers/parcels w can take at the same time.*

Definition 2 (Request)**.** *A request is defined as $r =< o_r, d_r, t_r, e_r, c_r >$, with an origin o_r, a destination d_r, a release time t_r, a deadline e_r, and a capacity c_r, where c_r is the number of passengers/parcels for request r. A request r can be completed if it is picked up after t_r and delivered before e_r by a worker.*

For ease of presentation, denote $R = \{r_1, r_2, ..., r_{|R|}\}$ as the set of requests assigned to w yet have not been completed.

Definition 3 (Route)**.** *Given a worker w and a request set R, a route of w is defined as $S_R = \langle l_0, l_1, l_2, ..., l_n \rangle$, which is a sequence of w's current location and all the origins and destinations of the requests in R, i.e. $l_0 = o_w$ and $l_i \in \{o_r | r \in R\} \cup \{d_r | r \in R\}$ for all $1 \le i \le n$. We use n to denote the number of locations in S_R except the current location of w.*

A route is feasible if the following constraints are satisfied.

- **Order Constraint.** $\forall r \in R$, o_r lies before d_r, *i.e.,* a request is picked up before delivered;
- **Deadline Constraint.** $\forall r \in R$, the worker w completes r before its deadline e_r, *i.e.,* all the assigned requests can be completed;
- **Capacity Constraint.** At any time, the total capacity of all requests that have been picked up but not delivered does not exceed the capacity of w.

Definition 4 (Flow Time)**.** *Given a worker w, a request set R and a feasible route S_R, the flow time of each request $r \in R$ is the duration between t_r and the time that r is delivered (denoted by $delv(r)$), i.e. $flw(r) = delv(r) - t_r$.*

Definition 5 (Insertion Operator)**.** *Given a worker w, a feasible route S_R, and a new request r', the insertion operator inserts $o_{r'}$ and $d_{r'}$ into S_R to obtain a new feasible route S_{R^+} ($R^+ = R \cup \{r'\}$). Depending on the specific applications, one of the following objective functions should be minimized.*

*(1) **Maximum flow time** of all the requests [18], [19], [20], [9], [10], i.e.* $\max_{r \in R^+}\{flw(r)\}$.

*(2) **Total travel time** of the worker [12], [2], [4], [8], or equivalently, **the delivery time of the last request**, i.e.* $\max_{r \in R^+}\{delv(r)\}$.

We make two remarks on the insertion operator.

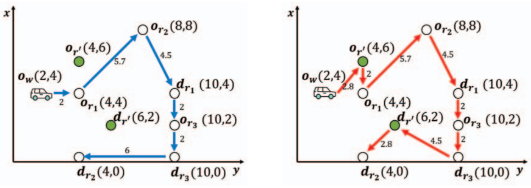
- For brevity,"insertion (i, j)" is used to denote the insertion of $o_{r'}$ after l_i and $d_{r'}$ after l_j.
- For convenience, we rewrite the two objective functions into a unified form as

$$\text{OBJ}(S_{R^+}) = \max_{r \in R^+}\{flw(r) + \alpha \cdot t_r\}, \quad (1)$$

where α is either 1 or 0. Note that

$$\text{OBJ}(S_{R^+}) = \begin{cases} \text{maximum flow time}, & \alpha = 0 \\ \text{total travel time}, & \alpha = 1 \end{cases} \quad (2)$$

The following example illustrates the insertion operator.



(a) Route before insertion. (b) Route after insertion.

Fig. 1: An example of insertion.

TABLE I: Information of requests.

request	release time t_r	deadline e_r	origin o_r	destination d_r	capacity c_r
r_1	0	25	$(4,4)$	$(10,4)$	1
r_2	0	37	$(8,8)$	$(4,0)$	1
r_3	0	33	$(10,2)$	$(10,0)$	1
r'	2	26	$(4,6)$	$(6,2)$	1

Example 1. *Suppose that on a ridesharing platform a driver w is serving three requests r_1, r_2 and r_3. At time 2, a new request r' arrives and we try to insert r' into the current route S_R of w. The origins and destinations of requests are shown in Fig. 1a, and their information is shown in Table I. At this time $S_R = \langle o_w, o_{r_1}, o_{r_2}, d_{r_1}, o_{r_3}, d_{r_3}, d_{r_2} \rangle$, where $o_w = (2,4)$. We account the travel time between locations to one decimal place. We also assume that the capacity of the worker c_w is 4 and the capacity of all the requests is 1.*

The new route S_{R^+} should satisfy the capacity constraint and deadline constraint, and keep the order of r_1-r_3's origins and destinations the same as in S_R. A feasible route after insertion is to insert $o_{r'}$ and $d_{r'}$ after o_w and d_{r_3} respectively, as shown in Fig. 1b. In the new route S_{R^+}, the flow time of four requests is $flw(r_1) = (2+2.8+2+5.7+4.5) - 0 = 17$, $flw(r_2) = (2+2.8+2+5.7+4.5+2+2+4.5+2.8) - 0 = 28.3$, $flw(r_3) = (2+2.8+2+5.7+4.5+2+2) - 0 = 21$, $flw(r') = (2+2.8+2+5.7+4.5+2+2+4.5) - 2 = 23.5$, respectively. Thus, the maximum flow time of the route is $\max\{17, 28.3, 21, 23.5\} = 28.3$; and the total travel time of the route is $2+2.8+2+5.7+4.5+2+2+4.5+2.8 = 28.3$.

III. Related Work

Ridesharing services first emerged in 1970s as a result of the oil crisis and has received increasingly attention due to the development of the mobile Internet, sharing economy and spatial crowdsourcing [22], [23], [24], [25]. The first research paper dates back to the pickup and delivery problem (*a.k.a.* dial-a-ride problem) proposed in 1975 [26], and has been extensively studied by the database, data mining, transportation science and operations research communities. For nearly 50 years, neither super-constant approximation algorithms nor hardness results are known for the dial-a-ride problem. Instead, *insertion* is widely used by various heuristic solutions to ridesharing [9], [10], [11], [27], [28], [12], [2], [4], [8] and is regarded as a basic operator in ridesharing [13], [14], [15]. Table II lists some of the most representative solutions to ridesharing based on insertion under different optimization objectives.

TABLE II: Time complexity for *insertion* in existing works.

Method and Reference	**Objective**	**Constraint**	**Time**	**Datasets for Evaluation**
exact [18]	max flow time	order, capacity	$O(n^4)$	synthetic
sequential insertion [11]	max flow time	order, capacity, deadline	$O(n^3)$	synthetic
adaptive insertion [9]	max flow time	order, capacity, deadline	$O(n^3)$	synthetic
large-scale insertion [10]	max flow time	order, capacity, deadline	$O(n^3)$	synthetic
clustering insertion [12]	total travel time	order, capacity, deadline	$O(n^3)$	synthetic
tshare [2], [3]	total travel time	order, capacity, deadline	$O(n^3)$	ridesharing
kinetic [4], [7], [6], [21] and single rider insertion [5]	total travel time	order, capacity, deadline	$O(n^2)$	ridesharing
pruneGreedyDP [8]	total travel time	order, capacity, deadline	$O(n)$	ridesharing
our approach in this paper	max flow time	order, capacity, deadline	$O(n)$	ridesharing &logistics
	total travel time	order, capacity, deadline	$O(n)$	

Alg. 1 illustrates a straightforward implementation of insertion. It enumerates all insertions and finds a route with minimal OBJ(S_{R^+}). Enumerating (i, j) (lines 2-3) is operated $O(n^2)$ times, while checking constraints and calculating the objective of the new route in lines 5-6 need $O(n)$ time. Hence its time complexity is $O(n^3)$, where n is the number of locations in S_{R^+}. We review the usage of insertion for ridesharing services of different optimization objectives below.

```
Algorithm 1: Brute Force Algorithm
  input : A worker $w$ with route $S_R$, a new request $r'$
  output: A new route $S_{R^+}$
1 $O^* \leftarrow \infty, S_{R^+} \leftarrow S_R$;
2 for $i \leftarrow 0$ to $n$ do
3     for $j \leftarrow i$ to $n$ do
4         $S \leftarrow$ insert $o_{r'}$ after $l_i$ and $d_{r'}$ after $l_j$ in $S_R$;
5         if $S$ is feasible and OBJ($S$) $< O^*$ then
6             $O^* \leftarrow$ OBJ($S$), $S_{R^+} \leftarrow S$;
7 return $S_{R^+}$;
```

Maximum flow time models the longest waiting time of the requests before they are served. It was first used to evaluate the inconvenience or dissatisfaction of the requests (passengers) in ridesharing services. To minimize the maximum flow time in ridesharing, Psaraftis [18] proposes an exact solution for this NP-hard problem. Since the solution takes exponential time, it is only applicable to small datasets (*e.g.* the total number of requests is fewer than 10 [18]). To handle larger n (*e.g.* the total number of requests is around 3000 [11]), Jaw *et al.* [11] propose a sequential insertion procedure, *i.e.,* sequentially inserting one request into the current route of the worker. The insertion procedure is widely used by many following papers [9], [10], [29]. Hame *et al.* [9] utilize insertion to adaptively solve the problem of [18]. For larger-scale datasets, Krumke *et al.* [10], [29] design a batch based framework where insertion can be directly used. Currently, it still takes $O(n)$ time to calculate the objective and check constraints [11], [9], [10] and the insertion to minimize the maximum flow time takes $O(n^3)$ time [11], [9], [10].

Total travel time indicates the preference of workers [30], *i.e.,* a worker usually wants to serve all requests in less time. To minimize the total travel time in ridesharing, Iochim *et al.* [12] cluster the nearest requests first and then construct the route for each worker by repeated insertion. They use the insertion procedure of [11] in $O(n^3)$ time and insert requests into different routes in parallel. Zheng *et al.* [2], [3] design a general framework that repeatedly executes an $O(n^3)$ insertion. Huang *et al.* [4] combines insertion and a trie-based data structure called kinetic such that the time complexity of insertion is reduced to $O(n^2)$. Kinetic is widely used by other proposals to minimize the total travel time of ridesharing [7], [6], [21]. Cheng *et al.* [5] propose another implementation of the $O(n^2)$ insertion called single rider insertion. Very recently, Tong *et al.* [8] further accelerate the insertion operator to minimize the total travel time to linear time.

In summary, insertion is the cornerstone of many existing solutions to ridesharing. Although insertion with linear time has been proposed for some special optimization objectives, the generic insertion operator still takes $O(n^3)$ time. With the increasing scale and real-time requirement of ridesharing services, the efficiency of the insertion operator has become a bottleneck. In this work, we accelerate the generic insertion operator to linear time.

IV. A Partition-Based Framework

In this section, we introduce a partition-based framework that leads to an $O(n^2)$ insertion operator. The key enabler is to check constraints and calculate the objective in $O(1)$ time using the partition framework rather than in $O(n)$ time as needed in the straightforward implementation of insertion in Alg. 1. We first explain the basic idea of partition in Sec. IV-A, based on which we devise an insertion operator of $O(n^2)$ time complexity using dynamic programming in Sec. IV-B.

A. Rationale of Partition

The key observation of the partition-based framework is that we can partition the requests (*i.e.,* R^+, including the current requests R and the new request r') into four *disjoint* sets and handle their constraints and objective *independently*.

The partition of requests is based on the concept of **detour**. A detour represents the increased travel time after inserting a new location compared with the travel time of the original route. Formally, the detour $det(k, p)$ of inserting origin/destination p between k-th location and $(k + 1)$-th location of route S_R can be calculated as below:

$$det(k, p) = dis(l_k, p) + dis(p, l_{k+1}) - dis(l_k, l_{k+1}).$$

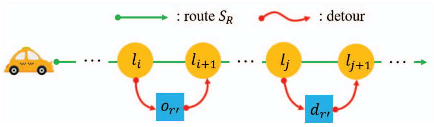


Fig. 2: An example of detour for insertion (i, j).

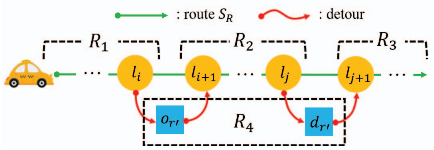


Fig. 3: An example of request partition $(i < j)$.

As shown in Fig. 2, given *insertion (i,j)*, we focus on two detours $det(i, o_{r'})$ and $det(j, d_{r'})$, *i.e.,* the detour of inserting $o_{r'}$ (the increased travel time from the i-th location and the $(i+1)$-th location) and the detour of inserting $d_{r'}$ (the increased travel time from the j-th location and the $(j+1)$-th location).

According to the difference in the impact of detours due to insertion (i, j) of a new request r', we can now partition all the requests into four disjoint sets (see Fig. 3).

- R_1 contains the requests whose destinations are before the i-th location (i included). All the requests in this set are not influenced by the detour of inserting $o_{r'}$ and $d_{r'}$.
- R_2 contains the requests whose destinations are between the i-th location (i excluded) and the j-th location (j included). All the requests in this set are influenced by detour of inserting $o_{r'}$.
- R_3 contains the requests whose destinations are after the j-th location (j excluded). All the requests in this set are influenced by detours of inserting $o_{r'}$ and $d_{r'}$.
- R_4 contains the new request r', which causes the detour.

With the above partition, Eq.(1) can be rewritten as

$$\text{OBJ}(S_{R+}) = \max\{mf_1, mf_2, mf_3, mf_4\} \tag{3}$$

where

$$mf_1 = \max_{r \in R_1}\{flw(r) + \alpha t_r\}$$
$$mf_2 = \max_{r \in R_2}\{flw(r) + \alpha t_r\}$$
$$mf_3 = \max_{r \in R_3}\{flw(r) + \alpha t_r\}$$
$$mf_4 = \max_{r \in R_4}\{flw(r) + \alpha t_r\}$$

Based on Eq.(3), we can also reformulate the framework of insertion as in Alg. 2. Specifically, for each pair of (i, j) for insertion (lines 1-2), we first check in line 3 if the capacity and deadline constraints are violated (Sec. IV-B1). If not, we calculate in line 4 the values of mf_1, mf_2, mf_3, mf_4. We finally calculate the objective in line 5 and update (i^*, j^*) which represents the best insertion locations in line 6.

Algorithm 2: Framework

```
  input : A worker $w$ with route $S_R$, a new request $r'$
  output: A new route $S_{R+}$
1 for $i \leftarrow 0$ to $n$ do
2     for $j \leftarrow i$ to $n$ do
3         Check the capacity and deadeline constraints;
4         Compute $mf_1$,$mf_3$,$mf_2$,$mf_4$ of insertion $(i, j)$;
5         OBJ $\leftarrow \max\{mf_1, mf_2, mf_3, mf_4\}$;
6         Update $(i^*, j^*)$ with $(i, j)$ according to OBJ;
```

B. Naive Dynamic Programming Based Insertion

This subsection introduces an $O(n^2)$ insertion operator based on the partition framework in Sec. IV-A. The key insight is that the partition allows pre-calculation of some variables such that checking constraints and calculating the objectives can be performed in $O(1)$ time rather than $O(n)$ as in Alg. 1. Table III summarizes the major notations.

*1) **Checking Capacity and Deadline Constraints**:* Recall that capacity constraint means that at any time the number of passengers/parcels carried by a worker cannot exceed the worker's capacity and the deadline constraint means all the requests picked by the worker should be delivered before the requests' deadlines. We next show how to check these two constraints in $O(1)$ with variables $pck(\cdot)$ and $slk(\cdot)$.

*1.1) **Checking Capacity Constraint.*** Given S_R, $pck(k)$ is defined as the number of requests picked but not delivered after w arrives at l_k. For all $0 \leq k \leq n$, $pck(k)$ can be pre-calculated in $O(n)$. With $pck(k)$ we can check the capacity constraint in $O(1)$ through Lemma. 1.

Lemma 1. *The capacity constraint will not be violated iff* $pck(i) \leq c_w - c_{r'}$ *and* $pck(j) \leq c_w - c_{r'}$.

Proof. On the one hand, to insert $o_{r'}$ after l_i, $pck(i)$ must be less or equal to $c_w - c_{r'}$, such that the worker's capacity is not exceeded. On the other hand, if $pck(j) > c_w - c_{r'}$, then for any $j' > j$, insertion (i, j') will also violate the capacity constraint. This is because during the traversal from l_i to $l_{j'}$, the number of passengers/parcels carried will exceed the capacity of worker at l_j. Thus we can break the enumeration of j in our framework. Based on the statement above, for the current insertion (i, j), if we find $pck(j) \leq c_w - c_{r'}$, the insertion will not violate the capacity constraint. □

*1.2) **Checking Deadline Constraint.*** Define $slk(k)$ as the maximum tolerable time for detour after l_k to satisfy the deadline constraint (*i.e.,* slack time). Thus,

$$slk(k) = \min\{slk(k+1), ddl(k+1) - arr(k+1)\}$$

where $arr(k)$ represents the arrival time to reach l_k in the original route and $ddl(k)$ represents the latest time to arrive at l_k without violating the deadline constraint. Specifically $ddl(k)$ can be calculated as

$$ddl(k) = \begin{cases} e_r - dis(o_r, d_r), & l_k \text{ is an origin} \\ e_r, & l_k \text{ is a destination.} \end{cases} \tag{4}$$

The value of $slk(k)$ for all $0 \leq k \leq n$ can be pre-calculated in $O(n)$ before enumerating all pairs (i, j) for insertion. With $slk(k)$ we can check the deadline constraint in $O(1)$. Specifically, three cases should be checked.

TABLE III: Summary of major notations.

Notation	Description
$dis(p_1, p_2)$	travel time between p_1 and p_2
$det(k, p)$	the detour time of inserting location p after l_k
$arr(k)$	arrival time of l_k
$mobj(i, j)$	maximum $flw(r) + \alpha t_r$ for requests whose destinations are between l_i and l_j in the original route
$slk(k)$	maximum tolerable time for detour after l_k
$pck(k)$	number of requests picked but not delivered after l_k

- Check whether any deadline constraint of all the existing requests is violated by inserting $o_{r'}$ after l_i, *i.e.,* whether $det(i, o_{r'}) \le slk(i)$;
- Check whether any deadline constraint of all the existing requests is violated by inserting $d_{r'}$ after l_j, *i.e.,* whether $dis(l_i, o_{r'}) + dis(o_{r'}, d_{r'}) + dis(d_{r'}, l_{i+1}) - dis(l_i, l_{i+1}) \le slk(i)$ when $i = j$ or $det(i, o_{r'}) + det(j, d_{r'}) \le slk(j)$ when $i < j$;
- Check whether the deadline constraint of the new request is violated, *i.e.,* whether $arr(i) + dis(l_i, o_{r'}) + dis(o_{r'}, d_{r'}) \le e_{r'}$ when $i = j$ or $arr(i) + det(i, o_{r'}) + dis(l_j, d_{r^*}) \le e_{r'}$ when $i < j$.

*2) **Calculating Objectives**:* We calculate mf_1, mf_2, mf_3 and mf_4 in $O(1)$ time during the enumeration of i and j as follows. Denote $mobj(i, j)$ as the maximum $flw(r) + \alpha \cdot t_r$ for any request whose destination is between the i-th location and the j-th location. Thus, it takes $O(n^2)$ time to pre-calculate $mobj(i, j)$ by enumerating i from 0 to n and j from i to n. Since the pre-calculation can be done in $O(n^2)$ time before enumerating all pairs (i, j) for insertion, it only takes $O(1)$ time to access $mobj(i, j)$ in the enumerations of insertion (i, j). We next show how to calculate mf_1, mf_2, mf_3 and mf_4 in $O(1)$ time in two cases: *(i)* $i < j$ and *(ii)* $i = j$.

When $i < j$, mf_1, mf_2, mf_3 and mf_4 can be calculated with the help of $mobj(i, j)$ in $O(1)$ time as follows.

- **Calculating mf_1:** As shown in Fig. 3, all the requests in R_1 (whose destination is before the i-th location) are not influenced by detour. Thus, mf_1 can be calculated as
$$mf_1 = mobj(0, i) \tag{5}$$
- **Calculating mf_2:** As shown in Fig. 3, all the requests in R_2 (whose destination is between the i-th and the j-th locations) are only influenced by the detour of inserting i. Specifically, $flw(r) + \alpha t_r$ of each request in R_2 would increase by $det(i, o_{r'})$. Thus mf_2 can be calculated as
$$mf_2 = det(i, o_{r'}) + mobj(i + 1, j) \tag{6}$$
- **Calculating mf_3:** As shown in Fig. 3, all the requests in R_3 (whose destination is after the j-th location) are influenced by the detours of inserting i and j. Specifically, $flw(r) + \alpha t_r$ of each request in R_3 would increase by $det(i, o_{r'}) + det(j, d_{r'})$. Thus mf_3 can be calcluated as
$$mf_3 = det(i, o_{r'}) + det(j, d_{r'}) + mobj(j + 1, n) \tag{7}$$
- **Calculating mf_4:** As shown in Fig. 3, R_4 only contains the new request r'. Intuitively, it would take $arr(j) + det(i, o_{r'})$ time to reach the j-th location, due to detour of inserting i. It will take another $dis(l_j, d_{r'})$ time to reach the destination of r'. Thus, we have
$$mf_4 = arr(j) + det(i, o_{r'}) + dis(l_j, d_{r'}) + (\alpha - 1)t_{r'} \tag{8}$$

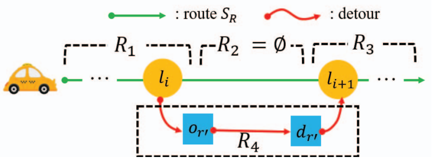


Fig. 4: An example of requests partition ($i = j$).

TABLE IV: Values of $mobj(\cdot, \cdot)$.

i \ j	0	1	2	3	4	5	6
0	0	0	0	14.2	14.2	18.2	24.2
1	-	0	0	14.2	14.2	18.2	24.2
2	-	-	0	14.2	14.2	18.2	24.2
3	-	-	-	14.2	14.2	18.2	24.2
4	-	-	-	-	0	18.2	24.2
5	-	-	-	-	-	18.2	24.2
6	-	-	-	-	-	-	24.2

When $i = j$, we calculate mf_1, mf_2, mf_3 and mf_4 in $O(1)$ time. The case when $i = j$ differs from the case when $i < j$ in two folds. *(i)* R_2 contains no requests when $i = j$. *(ii)* detour is calculated differently. Fig. 4 shows an example of the case when $i = j$. Accordingly, when $i = j$, mf_1, mf_2, mf_3 and mf_4 are calculated as follows.

- **Calculating mf_1:** mf_1 is still $mobj(0, i)$ since the requests in R_1 are not influenced by detour.
- **Calculating mf_2:** mf_2 is 0 because R_2 contains no requests when $i = j$.
- **Calculating mf_3:** Denote $det(i, r')$ as the detour when $i = j$. Then the $det(i, r')$ can be calculated as
$$dis(l_i, o_{r'}) + dis(o_{r'}, d_{r'}) + dis(d_{r'}, l_{i+1}) - dis(l_i, l_{i+1})$$
Thus mf_3 can be calculated as $det(i, r') + mobj(i + 1, n)$.
- **Calculating mf_4:** For mf_4, it takes $arr(i) + dis(l_i, o_{r'})$ time to reach $o_{r'}$ and then another $dis(o_{r'}, d_{r'})$ time to reach $d_{r'}$. Thus mf_4 can be calculated as
$$mf_4 = arr(i) + dis(l_i, o_{r'}) + dis(o_{r'}, d_{r'}) + (\alpha - 1)t_{r'} \tag{9}$$

In summary, after pre-calculating $mobj(\cdot, \cdot)$ in $O(n^2)$ time, it takes $O(1)$ time to calculate the objective in Eq.(3).

Example 2. *Back to the settings in Example. 1. Suppose that we want to calculate the maximum flow time of insertion* $(1, 5)$. *We pre-calculate the values of* $mobj(\cdot, \cdot)$ *as Table IV. Take* $i = 1$ *as an example. As* l_1 *and* l_2 *are the origins of* r_1 *and* r_2 *respectively, we have* $mobj(1, 1) = mobj(1, 2) = 0$. l_3 *is the destination of* r_1, *and* $flw(r_1) = 14.2$. *We have* $mobj(1, 3) = \max\{mobj(1, 2), 14.2\} = 14.2$. *In the same way we have* $mobj(1, 4) = mobj(1, 3) = 14.2$, $mobj(1, 5) = \max\{mobj(1, 4), 18.2\} = 18.2$ *and* $mobj(1, 6) = \max\{mobj(1, 5), 24.2\} = 24.2$.

Then we can calculate mf_1, mf_2, mf_3 *and* mf_4 *as follows. First the maximum flow time of requests in* R_1 *is* $mf_1 = mobj(0, 1) = 0$. *Since* $det(1, o_{r'}) = 0.8$, *the maximum flow time of requests in* R_2 *is* $mf_2 = det(1, o_{r'}) +$

Algorithm 3: Naive DP Algorithm

```
   input : A worker $w$ with route $S_R$, a new request $r'$
   output: A new route $S_{R^+}$
1  $S_{R^+} \leftarrow S_R, O^* \leftarrow \infty, i^* \leftarrow none, j^* \leftarrow none$;
2  Pre-calculate $pck(\cdot), slk(\cdot), mobj(\cdot,\cdot)$;
3  for $i \leftarrow 0$ to $n$ do
4      for $j \leftarrow i$ to $n$ do
5          if capacity constarint is violated then break ;
6          if deadline constraint is violated then
               continue ;
7          $mf_1, mf_2, mf_3, mf_4 \leftarrow$ calculate by
               Eq.(5)-Eq.(9);
8          $O \leftarrow \max\{mf_1, mf_2, mf_3, mf_4\}$;
9          if $O < O^*$ then
10             $O^* \leftarrow O, i^* \leftarrow i, j^* \leftarrow j$;
11 if $O^* < \infty$ then
12     $S_{R^+} \leftarrow$ insert $o_{r'}$ after $l_{i^*}$ and $d_{r'}$ after $l_{j^*}$ in $S_R$;
13 return $S_{R^+}$;
```

$mobj(2,6) = 25$ *(Eq.(6)). As for the requests in* R_3*, we have* $det(1, o_{r'}) = 0.8$ *and* $det(5, d_{r'}) = 1.3$*. Based on Eq.(7), the maximum flow time of requests in* R_3 *is* $mf_3 = det(1, o_{r'}) + det(5, d_{r'}) + mobj(6,6) = 0.8+1.3+24.2 = 26.3$*. To obtain the maximum flow time of requests in* R_4*, we first get* $arr(5) = 18.2$*,* $det(1, o_{r'}) = 2 + 4.5 - 5.7 = 0.8$ *and* $dis(l_5, d_{r'}) = 4.5$*. Substituting these results into Eq.(8), we have that the maximum flow time of requests in* R_4 *is* $mf_4 = arr(5) + det(1, o_{r'}) + dis(l_5, d_{r'}) = 23.5$.

Finally the maximum flow time for insertion $(1,5)$ *is* $\max\{0, 25, 26.3, 23.5\} = 26.3$.

*3) **Algorithm Details**:* Alg. 3 illustrates the procedure of the naive DP based insertion algorithm. In line 2, we pre-calculate $pck(\cdot), slk(\cdot), mobj(\cdot,\cdot)$ as in Sec. IV-B1 and Sec. IV-B2. While enumerating the pairs (i,j) for insertion in lines 3-4, we first check the capacity constraint in line 5. If the capacity constraint is violated, we can directly break the enumeration of j according to Lemma. 1. Then we check the deadline constraint in line 6. If all constraints are satisfied, we calculate mf_1, mf_2, mf_3, mf_4 according to Eq.(5)-Eq.(9) in line 7, and calculate the objective according to Eq.(3) in line 8. In lines 9-10, we update O^*, i^*, and j^* respectively. Finally we choose whether to return the new route S_{R^+} or the original route S_R based on O^* in lines 11-13.

Example 3. *Back to the settings in Example. 1. Table V summarizes the maximum flow time of each insertion* (i,j)*. Symbol "×" means that the insertion violates the constraints. The values of* $mobj(\cdot,\cdot)$ *have been pre-calculated in Table IV. Take* $i = 1$ *as an example. For each* j *from* 1 *to* 6*, we first check the capacity and deadline constraints of insertion* (i,j)*. The insertions* $(1,1)$ *to* $(1,5)$ *satisfy the constraints. We further calculate their maximum flow time as* $31.3, 31.3, 31.5, 31.5$ *and* 26.3 *respectively. From Table V we know that insertion* $(1,5)$ *leads to the minimum maximum flow time of requests.*

TABLE V: Values of OBJ(S_{R^+}).

i \ j	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	32.3	30.4	33.3	33.5	33.5	28.3	27.8(×)
1	-	31.3	31.3	31.5	31.5	26.3	25.8(×)
2	-	-	33.2	37	37(×)	31.8(×)	31.3(×)
3	-	-	-	37	42.2(×)	37(×)	36.5(×)
4	-	-	-	-	38.4(×)	39.2(×)	38.7(×)
5	-	-	-	-	-	34(×)	33.5(×)
6	-	-	-	-	-	-	32.7(×)

Complexity Analysis. In line 2, variable $pck(\cdot), slk(\cdot)$ can be pre-calculated in $O(n)$ time, but variable $mobj(\cdot,\cdot)$ needs $O(n^2)$ time and $O(n^2)$ space to be calculated. Checking constraints and obtaining OBJ(S_{R^+}) while enumerating i and j can be realized in $O(1)$ time. Hence the total time of lines 3-10 is $O(n^2)$. Lines 11-12 take $O(n)$ time. Thus, the naive DP based insertion has a time complexity of $O(n^2)$ and a space complexity of $O(n^2)$.

V. A Segment based DP Algorithm

In Sec. IV we propose a naive DP based insertion with $O(n^2)$ time complexity. In this section, we push the limit of the time complexity of the generic insertion operator to $O(n)$ time, which is the lower bound of the time complexity, *i.e.,* the time of scanning input. We first introduce a new equivalent expression of objective with only $O(n)$ time of pre-calculation in Sec. V-A, and then present key observations on the capacity and the deadline constraints in Sec. V-B. Based on the new expression and the observations, we introduce the basic idea of the segment based DP algorithm in Sec. V-C, and describe the detailed algorithm in Sec. V-D.

A. New Equivalent Expression of Objective

Basic Idea: In Eq.(3), we calculate the objective OBJ(S_{R^+}) as $\max\{mf_1, mf_2, mf_3, mf_4\}$ when enumerating i and j. According to associative law, we can combine the objective in the following orders: *(i)* First combine mf_2 and mf_3 as com_1, *i.e.,* $com_1 = \max\{mf_2, mf_3\}$; *(ii)* Then combine mf_1 (denoted by com_2), *i.e.,* $com_2 = \max\{mf_1, mf_2, mf_3\} = \max\{mf_1, com_1\}$; *(iii)* Finally combine mf_4, *i.e.,* OBJ(S_{R^+}) $= \max\{com_2, mf_4\}$.

The naive DP insertion needs to pre-calculate a two dimensional array $mobj(i,j)$, which takes $O(n^2)$ time. By following the above order, we only need a column $(j = n)$ of this array, *i.e.,* $mobj(i,n)$. We first explain the calculation based on this new expression when $i < j$ as follows.

- **Calculating** com_1: We first separate the common term $det(i, o_{r'})$ from $\max\{mf_2, mf_3\}$ as $det(i, o_{r'}) + \max\{mobj(i+1, j), det(j, d_{r'}) + mobj(j+1, n)\}$. Then we focus on $mobj(i+1, j)$ in the second term because it cannot be calculated from the one dimentional array $mobj(\cdot, n)$. The trick is to combine an additional term $mobj(j+1, n)$ into the second term as $\max\{mobj(i+1, j), mobj(j+1, n), det(j, d_{r'}) + mobj(j+1, n)\}$. Combining $mobj(j+1, n)$ causes no change in the maximum because $mobj(j+1, n)$ is always no larger than

$det(j, d_{r'}) + mobj(j+1, n)$. Further note that the maximum between $mobj(i+1, j)$ and $mobj(j+1, n)$ is $mobj(i+1, n)$. Thus com_1 can be calculated as

$$det(i, o_{r'}) + \max\{mobj(i+1, n), det(j, d_{r'}) + mobj(j+1, n)\} \tag{10}$$

- **Calculating** com_2: Since $mf_1 = mobj(0, i)$, we have $com_2 = \max\{mobj(0, i), com_1\}$. Based on Eq.(10), $mobj(i+1, n)$ is no larger than com_1. Thus we can safely combine $mobj(i+1, n)$ into com_2 as:

$$\begin{aligned} com_2 &= \max\{mobj(0, i), mobj(i+1, n), com_1\} \\ &= \max\{mobj(0, n), com_1\} \end{aligned} \tag{11}$$

- **Calculating Objectives**: To calculate $\mathrm{OBJ}(S_{R^+}) = \max\{com_2, mf_4\} = \max\{mobj(0, n), com_1, mf_4\}$, we first calculate the last two terms and combine with $mobj(0, n)$. Since both com_1 and mf_4 contain $det(i, o_{r'})$, we extract it from $\{com_1, mf_4\}$ as follows.

$$\begin{aligned} det(i, o_{r'}) + \max\{&mobj(i+1, n), det(j, d_{r'}) + mobj(j+1, n), \\ &arr(j) + dis(l_j, d_{r'}) + (\alpha - 1)t_{r'}\} \end{aligned}$$

Denote $par(j)$ as the terms only related to j as follows.

$$\begin{aligned} par(j) = \max\{&det(j, d_{r'}) + mobj(j+1, n), \\ &arr(j) + dis(l_j, d_{r'}) + (\alpha - 1)t_{r'}\} \end{aligned} \tag{12}$$

Finally, we can rewrite $\mathrm{OBJ}(S_{R^+})$ in Eq.(3) as:

$$\max\Big\{mobj(0, n), det(i, o_{r'}) + \max\{mobj(i+1, n), par(j)\}\Big\} \tag{13}$$

When $i = j$, we use a similar way to reduce the time of pre-calculation. Specifically, since $mf_2 = 0$, we can safely combine $mobj(i+1, n)$ into the objective as

$$\begin{aligned} &\max\{mobj(0, i), mobj(i+1, n), det(i, r') + mobj(i+1, n)\} \\ &= \max\{mobj(0, n), det(i, r') + mobj(i+1, n)\} \end{aligned}$$

When we enumerate i, $det(i, o_{r'})$ is constant. It takes $O(1)$ time to calculate the objective and check constraints when $i = j$. Thus it takes $O(n)$ time in total to calculate the objective and check the constraints when $i = j$.

When $i < j$, even if i is fixed (enumerate), we still need to check each j ($> i$) in the naive DP insertion. As next, we introduce observations on the constraints, which help filter j that satisfies the capacity and the deadline constraints.

B. Observations on Capacity and Deadline Constraints

Observation on capacity constraint: In the naive DP insertion (Alg. 3), we can safely break the inner loop of j according to Lemma. 1. For each i, let $brk(i)$ be the value of j when it breaks the inner loop. It indicates that the capacity constraint is not violated for any j larger than i but not exceeds the breaking point $brk(i)$, *i.e.,* $i < j < brk(i)$. After comparing the inner loop for adjacent i, *i.e.,* $j \in (i, brk(i))$, we have the following observation.

Lemma 2. *(1) If the capacity constraint is violated when inserting $o_{r'}$ after the i-th location, i.e. $pck(i) > c_w - c_{r'}$, range $(i, brk(i))$ is empty. (2) Otherwise, the value of brk(i) is the same as brk(i + 1).*

Proof. (1) According to Lemma. 1, $pck(i) > c_w - c_{r'}$ indicates that the capacity constraint is violated when inserting $o_{r'}$ after i-th. Thus, the inner loop in Alg. 3 will break and range $(i, brk(i))$ is empty because $brk(i) = i$.

(2) First consider the case when $pck(i+1) > c_w - c_{r'}$. In this case $(i, brk(i))$ is empty according to (1). If $pck(i+1) \le c_w - c_{r'}$, insertion (i, j) satisfies the capacity constraint for all $j \in (i+1, brk(i))$. According to Lemma. 1, $brk(i) \le brk(i+1)$. Since we use $brk(i+1)$ to denote the rightmost j which satisfies the capacity constraint, *i.e.,* $brk(i+1) \ge brk(i)$. Thus, we have $brk(i) = brk(i+1)$. □

Observation on deadline constraint: According to the deadline constraints in Sec. IV-B1, we have the following observation, as illustrated in Lemma. 3.

Lemma 3. *Let $thr(j)$ be a threshold of j,*

$$thr(j) = \min\{slk(j) - det(j, d_{r'}), e_{r'} - arr(j) - dis(l_j, d_{r'})\}. \tag{14}$$

Assume the deadline constraint of existing requests is not violated by inserting o_r after the i-th location. Insertion (i, j) would satisfy the deadline constraint, iff the threshold of j is no less than detour of inserting i, i.e. $thr(j) \ge det(i, o_{r'})$.

Proof. According to Sec. IV-B1, the deadline constraint will not be violated iff
(1) $det(i, o_{r'}) \le slk(i)$;
(2) $det(i, o_{r'}) + det(j, d_{r'}) \le slk(j)$;
(3) $arr(j) + det(i, o_{r'}) + dis(l_j, d_{r'}) \le e_{r'}$.
The first condition (*i.e.,* $det(i, o_{r'}) \le slk(i)$) can be checked directly while enumerating i. Assume this condition is true. We rewrite the remaining two conditions as

$$\begin{aligned} det(i, o_{r'}) &\le slk(j) - det(j, d_{r'}), \\ det(i, o_{r'}) &\le e_{r'} - arr(j) - dis(l_j, d_{r'}). \end{aligned}$$

By defining $thr(j)$ as above, we get our conclusion. □

In summary, the first observation (from the capacity constraint) determines the range of j, *i.e.,* $i < j < brk(i)$. The second observation (from the deadline constraint) shows that only some of such j would satisfy both constraints, *i.e.,* those j whose threshold $thr(j)$ are no less than $det(i, o_{r'})$. In the coming subsections, as we enumerate i, we aim to calculate the minimum objective from such j more efficiently, *i.e.,*

$$\min_{\substack{i<j<brk(i) \\ thr(j) \ge det(i, o_{r'})}} \mathrm{OBJ}(S_{R^+}) \tag{15}$$

C. Segment based Optimization

Basic Idea: If we enumerate i, by utilizing data structure like segment tree [31], we can directly query the optimal j and the corresponding objective (*i.e.,* Eq.(15)). Next we explain in detail how to utilize the segment tree to accelerate constraint checking and objective calculation.

To efficiently filter those j satisfying the **deadline constraint** (*i.e.,* $thr(j) \ge det(i, o_{r'})$), we can construct a segment tree according to $thr(j)$. As i is fixed, then $det(i, o_{r'})$ is constant. By querying the segment $[det(i, o_{r'}), \infty)$, we filter those j satisfying the deadline constraints.

To efficiently calculate the **minimum objective** (*i.e.,* Eq.(15)), we store $par(j)$ (only related to j) as the value of

Algorithm 4: Segment based DP Algorithm

```
input : A worker $w$ with route $S_R$, a new request $r'$
output: A new route $S_{R+}$
1  $S_{R+} \leftarrow S_R, O^* \leftarrow \infty, i^* \leftarrow none, j^* \leftarrow none$;
2  Pre-calculate $pck(\cdot), slk(\cdot), thr(\cdot), mobj(\cdot, n)$;
3  Construct a segment tree ST;
4  for $i \leftarrow 0$ to $n$ do
5      Handle the case when $i = j$;
6  for $i \leftarrow n-1$ to $0$ do
7      Update leaf node $thr(i+1)$ with $par(i+1)$ in ST;
8      if $pck(i+1) > c_w - c_{r'}$ then
9          Invalidate ST;
10     if $pck(i) \leq c_w - c_{r'}$ and $det(i, o_{r'}) \leq slk(i)$ then
11         Query the minimum $par(j)$ from segment
             $[det(i, o_{r'}), \infty)$ in ST;
12         $O \leftarrow$ calculate objective according to Eq.(16);
13         if $O < O^*$ then
14             $O^* \leftarrow O, i^* \leftarrow i, j^* \leftarrow j$;
15 if $O^* < \infty$ then
16     $S_{R+} \leftarrow$ insert $o_{r'}$ after $l_{i^*}$ and $d_{r'}$ after $l_{j^*}$ in $S_R$;
17 return $S_{R+}$;
```

each leaf node in the tree. Thus, we can efficiently query the minimum value of $par(j)$ among previously filtered positions. As a result, we can efficiently calculate Eq.(15) for a fixed i. Specifically, the terms in Eq.(13) like $mobj(0, n)$, $det(i, o_{r'})$, $mobj(i+1, n)$ are constant for a fixed i. Substituting Eq.(13) into Eq.(15), we have:

$$\max\Big\{mobj(0, n), det(i, o_{r'}) + mobj(i+1, n), \qquad (16)$$
$$det(i, o_{r'}) + \min_{\substack{i<j<brk(i)\\ thr(j)\geq det(i, o_{r'})}} \{par(j)\}\Big\}$$

To maintain the positions of j from $(i, brk(i))$ which satisfy the **capacity constraint**, we either invalidate the segment tree or update the segment tree when enumerating i. Specifically, if inserting $o_{r'}$ after i-th location violates the capacity constraint (*i.e.,* Lemma. 2 (1)), we mark the tree as invalid; otherwise (*i.e.,* Lemma. 2 (2)), we update the tree. This way, both operations are efficient on the segment tree.

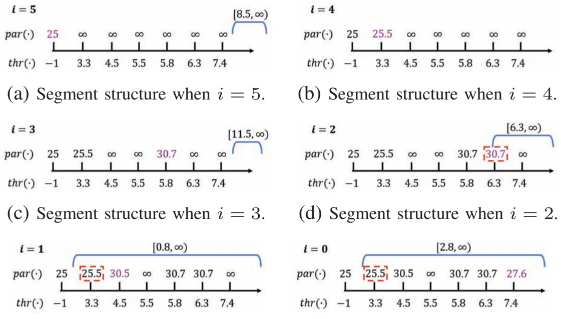
In summary, by utilizing segment tree and enumerating i, we can calculate the optimal j and the corresponding objective (Eq.(16)) efficiently.

D. Algorithm Details

Alg. 4 illustrates the process of the segment based DP insertion algorithm. In line 2, we pre-calculate $pck(\cdot)$, $slk(\cdot)$, $thr(\cdot)$, $mobj(\cdot, n)$ as in Sec. IV-B. In line 3, we construct a segment tree **ST**. Next, we handle the case when $i = j$ in lines 4-5. We enumerate i from $n-1$ to 0 in line 6. For a fixed i, we first update the **ST** with value $par(i+1)$ at $thr(i+1)$ in **ST** in line 7. In lines 8-9, we invalidate the **ST** if the capacity constraint of $i+1$ is violated. In line 10, we check whether inserting $o_{r'}$ after the i-th violates the capacity and deadline constraints. If not, we query the optimal j and the minimum value among segment $[det(i, o_{r'}), \infty)$ in line 11. In line 12, we calculate the current objective value according to Eq.(16). In lines 13-14, we update O^*, i^* and j^* according to the current objective O. Finally, we choose whether to return the new route S_{R+} or the original route S_R in lines 15-17.

TABLE VI: Values of notations in Example. 4.

$index$	$0(o_w)$	$1(o_{r_1})$	$2(o_{r_2})$	$3(d_{r_1})$	$4(o_{r_3})$	$5(d_{r_3})$	$6(d_{r_2})$
$thr(\cdot)$	5.5	7.4	4.5	6.3	5.8	3.3	-1
$mobj(\cdot, 6)$	24.2	24.2	24.2	24.2	24.2	24.2	24.2
$par(\cdot)$	29.5	27.6	30.5	30.7	30.7	25.5	25


i = 5
[8.5, ∞)
par(·) 25 ∞ ∞ ∞ ∞ ∞ ∞
thr(·) −1 3.3 4.5 5.5 5.8 6.3 7.4
i = 4
par(·) 25 25.5 ∞ ∞ ∞ ∞ ∞
thr(·) −1 3.3 4.5 5.5 5.8 6.3 7.4
i = 3
[11.5, ∞)
par(·) 25 25.5 ∞ ∞ 30.7 ∞ ∞
thr(·) −1 3.3 4.5 5.5 5.8 6.3 7.4
i = 2
[6.3, ∞)
par(·) 25 25.5 ∞ ∞ 30.7 30.7 ∞
thr(·) −1 3.3 4.5 5.5 5.8 6.3 7.4
i = 1
[0.8, ∞)
par(·) 25 25.5 30.5 ∞ 30.7 30.7 ∞
thr(·) −1 3.3 4.5 5.5 5.8 6.3 7.4
i = 0
[2.8, ∞)
par(·) 25 25.5 30.5 ∞ 30.7 30.7 27.6
thr(·) −1 3.3 4.5 5.5 5.8 6.3 7.4

(a) Segment structure when $i = 5$. (b) Segment structure when $i = 4$.

(c) Segment structure when $i = 3$. (d) Segment structure when $i = 2$.

(e) Segment structure when $i = 1$. (f) Segment structure when $i = 0$.

Fig. 5: Segment structures in Example. 4.

Note that in real-world ridesharing services, the time period from the pickup to the delivery of a request is usually bounded and reasonably short. Hence in practice, for a given i, the number of j which may lead to a feasible insertion is bounded by a constant and these positions can be maintained by dynamic structures, *e.g.* fenwick tree (dynamic version), whose construction time is $O(n)$ and whose maintain time is $O(1)$.

Example 4. *Back to the settings in Example. 1. We aim to find the minimum maximum flow time of requests. Table VI summarizes the values after pre-calculation. We have obtained the values of $mobj(\cdot, 6)$ as shown in Table IV. The values of $thr(\cdot)$ and $par(\cdot)$ are generated by their definitions in Sec. V-B and Sec. V-C, respectively. For example, $thr(0)$ is the minimum of $slk(0) - det(0, d_{r'})$ and $e_{r'} - arr(0) - dis(l_0, d_{r'})$, which is* 5.5. *Also, $par(0) = \max\{det(0, d_{r'}) + mobj(1, 6), arr(0) + dis(l_0, d_{r'}) + (\alpha - 1)t_{r'}\} = 29.5$.*

Fig. 5 shows the data structure based on $thr(\cdot)$ and its stored information while enumerating i. In each figure the values over the axis record the values of $par(k)$ for k from i to $n = 6$ and the value in purple represents the newly inserted one. When $i = 5$, $par(6) = 25$ and we update 25 *in the structure, as shown in Fig. 5a. Then we query the optimal $par(j)$ from the segment $[det(5, o_{r'}), \infty) = [8.5, \infty)$ (blue curve in Fig. 5a). The query returns ∞ (which means such j does not exist) and we do not update the optimal route O^*. For $i = 4$, $par(5) = 25.5$ is updated. Observe that $det(4, o_{r'}) > slk(4)$, we skip the query. For $i = 3$, we update $par(4) = 30.7$ and the query returns ∞, which is similar to the case of $i = 5$. When $i = 2$, $par(3) = 30.7$ is updated. The query from segment $[det(2, o_{r'}), \infty) = [6.3, \infty)$*

returns the optimal $par(j) = 30.7$ *with* $j = 3$. *In this case the maximum flow time is* $\max\{mobj(0,6), det(2, o_{r'}) + \max\{mobj(3,6), 30.7\}\} = 37$ *and the corresponding optimal insertion is* $(2,3)$. *For the case* $i = 1$, 30.5 *is updated and the query from segment* $[det(1, o_{r'}), \infty)$ *returns the optimal* $par(j) = 25.5$ *with* $j = 5$. *In this case the segment* $(1,5)$ *leads to the maximum flow time* 26.3. *Similarly the case* $i = 0$ *leads to the maximum flow time* 28.3. *Finally we have the minimum maximum flow time is* 26.3 *with the optimal insertion* $(1,5)$.

Complexity Analysis. We analyze the complexity of Alg. 4 with two implementations, *segment tree* and *fenwick tree*.

Complexity of Alg. 4 with Segment Tree Implementation. Pre-calculations in line 2 take $O(n)$ time. In line 3, it takes $O(n \log n)$ to construct a segment tree **ST**. In lines 4-5, it takes $O(n)$ to handle the case when $i = j$. When enumerating i in lines 6-14, each operation (update in line 7, invalidation in line 9 and query in line 11) on the segment tree takes at most $O(\log n)$ time. Lines 8, 10, 12-14 take $O(1)$ time. Lines 15-16 take $O(n)$ time. Hence the total time complexity of Alg. 4 implemented with a segment tree is $O(n \log n)$. Since the pre-calculation only consumes $O(n)$ space and the size of a segment tree is also $O(n)$, the total space complexity of Alg. 4 implemented with a segment tree is $O(n)$.

Complexity of Alg. 4 with Fenwick Tree Implementation. Compared with the segment tree implementation, we construct a fenwick tree (dynamic version) in $O(n)$ in line 3. With the fenwick tree implementation, the update (line 7), validation (line 9) and query (line 11) operations take $O(1)$ time. The time complexity of the other lines is the same as that of Alg. 4 with segment tree implementation. Finally, the time complexity of Alg. 4 with fenwick tree implementation is $O(n)$. As the size of fenwick tree is also $O(n)$, the total space complexity is the same as that of Alg. 4 with segment tree implementation, which is $O(n)$.

VI. Experimental Study

This section presents the evaluation of our algorithms.

A. Experimental Setup

Datasets. We experiment with two real datasets (Table VII).

The first dataset [32] (denoted by ***Taxi***) is the trip records of yellow and green taxis in New York City. We choose the data on April 09, 2016, which has the largest number of requests (2nd row in Table VII) in a single day. The dataset is pre-processed as follows. There are $10,000$ workers whose origins are uniformly generated on the road network in ***Taxi***. The origins and the destinations of requests are mapped to the closest vertex in the road network of New York City extracted from [33] and the speed on an edge of the road network is set to be 80% of the maximum legal speed limit. As there is no capacity information in the dataset, we generate the capacities of the workers by a Gaussian distribution whose mean varies from 3 to 20 (2nd row of Table VIII). Considering that short trips dominate in the requests [34], we vary the value of $e_r - t_r$ from 10 to 30, which is the period from release time to deadline of a request (4th row of Table VIII). To test the algorithms with different amounts of requests, we extract the first 20k to 100k requests for evaluation (3rd row of Table VIII). To test the scalability of the algorithms, we extract the first 100k to 500k requests for evaluation (5th row of Table VIII). Note the number of requests is not the length n of a route. The default settings are marked in bold.

TABLE VII: Statistics of datasets.

Dataset	Space	#(Requests)	#(Vertices)	#(Edges)
Taxi	Road network	517,100	807,795	2,100,632
Logistics	Euclidean space	345,849	12,487	Connected between any vertex

TABLE VIII: Parameter settings.

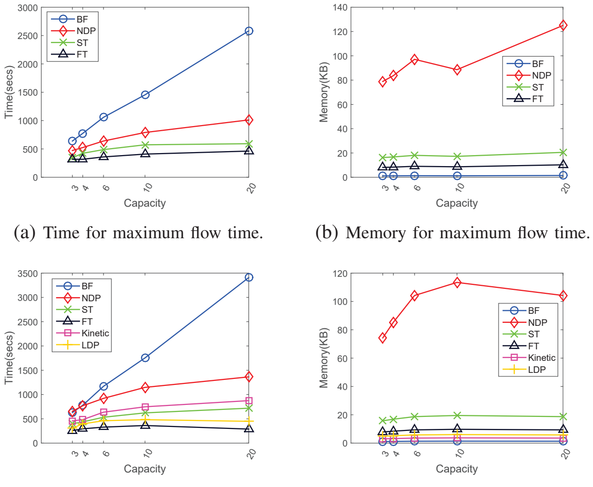
Parameters	Settings
Capacity c_w	***Taxi***: $3, \mathbf{4}, 6, 10, 20$ ***Logistics***: $80, 100, \mathbf{120}, 140, 160$
Number of requests	***Taxi***: 20k,40k,**60**k,80k,100k ***Logistics***: 2k,4k,**6**k,8k,10k
Time period from release time to deadline $e_r - t_r$ (minute)	***Taxi***: $10, \mathbf{15}, 20, 25, 30$ ***Logistics***: original deadline information
Scalability	***Taxi***: $100k$,200k,300k,400k,500k ***Logistics***: 60k,120k,180k,240k,300k

The second dataset (denoted by ***Logistics***) comes from Cainiao [35] , a well-known logistics platform in China, and is published as the dataset of the parcel delivery contest in Tianchi [36], an AI developer community. The dataset contains the origins and the destinations as well as the deadline information of the parcels (requests) in a day in Shanghai (3rd row in Table VII). We pre-process ***Logistics*** in a similar way to ***Taxi*** and the parameter settings are shown in Table VIII. In total 150 workers ($5,000$ for scalability) are uniformly generated on the euclidean space to deliver the requests. The only difference is that we directly use the deadline information of requests in ***Logistics***.

The experiments are conducted on a server with 40 Intel(R) Xeon(R) E5 2.30GHz processors with hyper-threading enabled and 128GB memory. All of the algorithms are implemented in GNU C++. Each experiment is repeated 10 times and we show the average results.

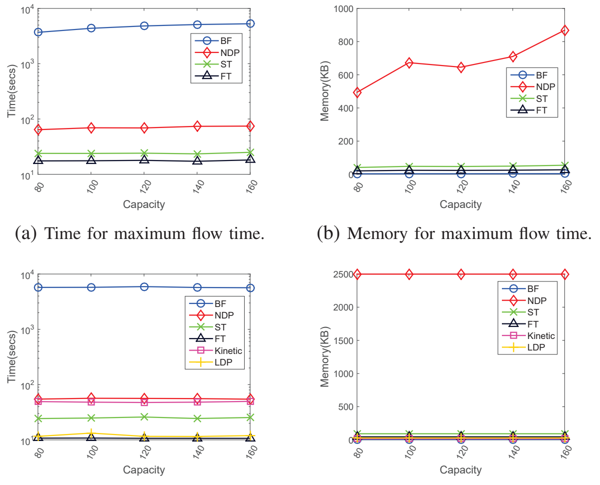
Compared Algorithms. We evaluate the performance of the following algorithms.

- **BF** (Brute Force) is an $O(n^3)$ insertion operator that enumerates the origin and the destination to find the optimal insertion (Alg. 1).
- **NDP** (Naive DP) is an $O(n^2)$ insertion operator that enumerates the origin and the destination to find the optimal insertion (Alg. 3).
- **ST** (Segment based DP with segment tree implementation) only enumerates the origin and finds the optimal insertion using the segment tree (Alg. 4 implemented with segment tree). Its time complexity is $O(n \log n)$.
- **FT** (Segment based DP with fenwick tree implementation) only enumerates the origin and finds the optimal insertion using the fenwick tree (Alg. 4 implemented with fenwick tree). Its time complexity is $O(n)$.
- **Kinetic** [4] is a widely used $O(n^2)$ dynamic programming based insertion operator for minimizing the total



(a) Time for maximum flow time.
(b) Memory for maximum flow time.
(c) Time for total travel time.
(d) Memory for total travel time.

Fig. 6: Results of varying capacity of workers on ***Taxi***.



(a) Time for maximum flow time.
(b) Memory for maximum flow time.
(c) Time for total travel time.
(d) Memory for total travel time.

Fig. 7: Results of varying capacity of workers on ***Logistics***.

travel time.

- **LDP** [8] is the state-of-the-art dynamic programming based insertion operator for minimizing the total travel time. Its time complexity is $O(n)$.

Note that **LDP** [8] and **Kinetic** [4] are only applicable in minimizing the total travel time. Hence we exclude these two algorithms when comparing the performance to minimize the maximum flow time.

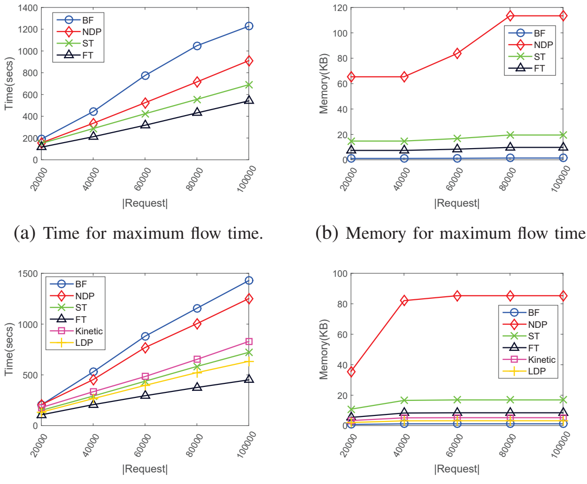
Metrics. We integrate the above insertion algorithms into a widely used route planning solution to dynamic ridesharing [2], [4], [8]. Upon arrival of a new request, the solution inserts a new request to all possible workers who can pick up the request using the insertion operator and greedily returns the best insertion locations and the corresponding worker. As previous works like [37], [38], we compare the memory and time costs of such a route planning solution with different implementations of the insertion operator on real-world large-scale datasets. Specifically, we report the *maximum memory cost during insertion* and the *total time of all the insertions on each dataset* when using different insertion operators for ridesharing. Note that the number of insertion operators called by the greedy solution is the same for different insertion algorithms. Hence the total memory and time costs can reflect the performance of these compared algorithms.

B. Experimental Results

Impact of Capacity of Workers. Fig. 6 and Fig. 7 show the results of varying the capacity of workers on ***Taxi*** and ***Logistics***, respectively. FT has the shortest running time in both objectives, which is up to 6.4 and 290.8 times faster than the others on ***Taxi*** and ***Logistics***, respectively. Specifically, when minimizing the total travel time, FT is even slightly faster than LDP, although both algorithms have a linear time complexity. With the increase in the capacity of workers, the time cost of BF grows and the time costs of the other algorithms remain stable on ***Taxi***. On ***Logistics***, the time costs of all the algorithms are stable. This may be because with a small capacity (on ***Taxi***) the length of routes is dominated by the capacity while when the capacity increases, the length of routes is limited by the number of the requests. The memory costs of all the algorithms except NDP remain almost the same when varying the capacity of workers, while BF consumes the least memory. Note that the memory cost of NDP changes in a similar trend to that of ST and FT but is more notable, due to its $O(n^2)$ space complexity. ST and FT only consume slightly more memory than BF (less than 80 KB), which validates the memory efficiency of these two algorithms.

Impact of Number of Requests. Fig. 8 and Fig. 10 show the results of varying the number of requests on ***Taxi*** and ***Logistics***, respectively. FT still outperforms the other algorithms in terms of the running time when minimizing the maximum flow time, *i.e.,* 2.2 and 998.1 times faster than BF on ***Taxi*** and ***Logistics***, respectively. When minimizing the total travel time, FT is faster than LDP on ***Taxi*** and is as fast as LDP on ***Logistics*** and both of them are faster than the other algorithms. With the increasing number of requests, the time costs of all the algorithms increase on both ***Taxi*** and ***Logistics***. This is because with the increase of number of requests, workers tend to obtain a longer route and thus need longer time to complete the route. As for memory, BF still has the lowest memory consumption. NDP performs the worst as it consumes $O(n^2)$ memory to store the variables. The gap of memory cost among algorithms (except NDP) is marginal (less than 0.1 MB).

Impact of Deadline of Requests. Fig. 9 shows the results of varying the deadline on ***Taxi***. The horizontal axis represents the values of $e_r - t_r$. FT is again the fastest among all the algorithms, which is up to 3.7 times faster. With the increase of $e_r - t_r$, the time costs of all the algorithms increase, while those of FT and LDP increase slower than BF, Kinetic, ST and NDP. This is because with a larger deadline, more requests



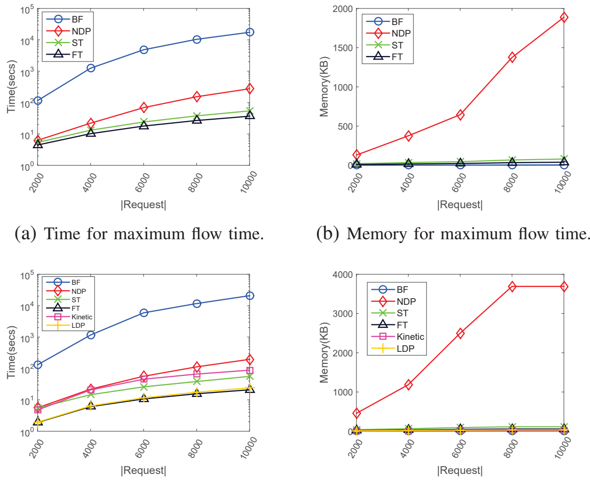
(a) Time for maximum flow time.

(b) Memory for maximum flow time.

(c) Time for total travel time.

(d) Memory for total travel time.

Fig. 8: Results of varying # of requests on ***Taxi***.



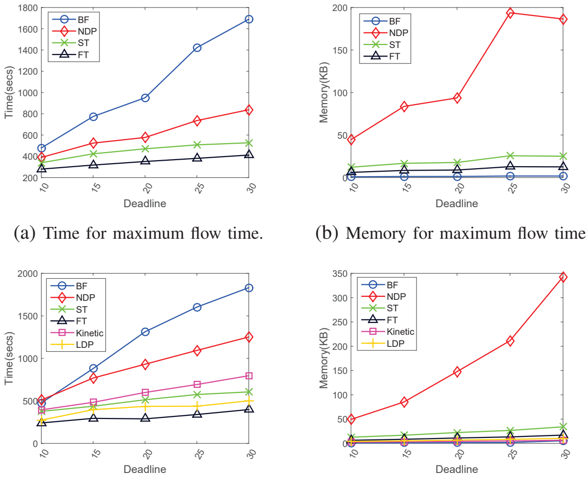
(a) Time for maximum flow time.

(b) Memory for maximum flow time.

(c) Time for total travel time.

(d) Memory for total travel time.

Fig. 10: Results of varying # of requests on ***Logistics***.



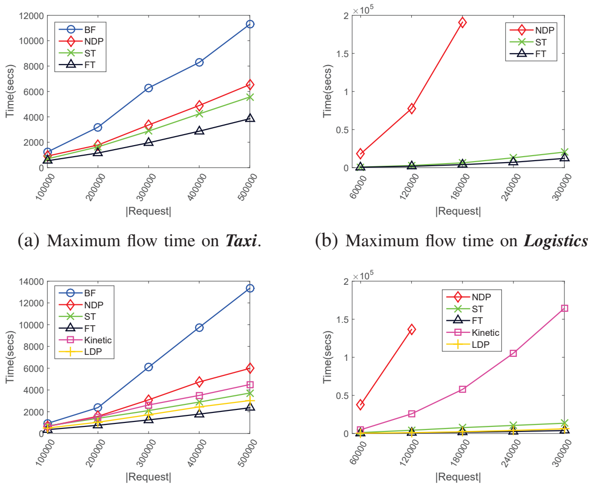
(a) Time for maximum flow time.

(b) Memory for maximum flow time.

(c) Time for total travel time.

(d) Memory for total travel time.

Fig. 9: Results of varying $e_r - t_r$ on ***Taxi***.



(a) Maximum flow time on ***Taxi***.

(b) Maximum flow time on ***Logistics***.

(c) Total travel time on ***Taxi***.

(d) Total travel time on ***Logistics***.

Fig. 11: Results of scalability test on # of requests.

can be inserted into the route, and FT and LDP have a lower time complexity. The memory costs of all the algorithms remain stable with the increase of the deadlines of requests except NDP. Again BF has the lowest memory costs, while the memory costs of ST and FT are only slightly higher (less than 20 KB more memory). NDP consumes the most memory.

Scalability. Fig. 11 shows the experimental results on scalability. We omit the memory cost due to limited space. On ***Logistics***, BF and NDP fail to terminate in two days so we omit the results of these experiments. FT runs faster than the other algorithms in both objectives. With the increase of requests, the time costs of all the algorithms increase while the time costs of FT and LDP have a lower increasing speed. The results show that our proposed algorithms, ST and FT, are fit for large-scale datasets.

Comparison between Datasets. Comparing the results on ***Taxi*** (Fig. 6, Fig. 8, Fig. 9 and Fig. 11) and ***Logistics*** (Fig. 7, Fig. 10 and Fig. 11), we have the following observations.

- On both datasets FT outperforms the other algorithms in terms of running time, except for Fig. 10c where LDP runs as fast as FT.
- All the algorithms consume more memory on ***Logistics*** (40-2500 KB) than on ***Taxi*** (10-140 KB). This may be because on ***Logistics*** requests have a larger capacity than on ***Taxi***. This leads to more feasible insertion locations for each request and increases the memory cost.

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

- Insertion with the straightforward implementation (*i.e.,* BF) is impractical for real-world dynamic ridesharing applications (more than 24 hours on ***Logistics***).
- Our algorithms NDP, ST and FT are 1.5 to 6.4 times

faster than BF on ***Taxi***, and are 4.3 to 998.1 times faster than BF on ***Logistics***.

- Our ST algorithm is up to 2.0 times and 5.1 times faster than NDP on ***Taxi*** and ***Logistics***, while our FT algorithm is even faster, *i.e.,* 2.9 to 7.6 times faster than NDP.
- For the objective to minimize the total travel time, our algorithm FT runs faster than LDP, the state-of-the-art insertion to minimize the total travel time, in most of the experiments. Note that our FT algorithm also runs the fastest when minimizing the maximum flow time.
- The memory costs of ST and FT are only slightly larger (within 0.1 MB) than the memory usage of BF.

VII. Conclusion

In this paper, we study the insertion operator, a widely used core operation in real-world dynamic ridesharing applications. A straightforward implementation of the insertion operator takes $O(n^3)$ time to obtain the optimal insertion locations. We propose a partition framework and devise a novel dynamic programming based insertion operator to reduce the time complexity of the generic insertion operator from $O(n^3)$ to $O(n^2)$. Leveraging fenwick tree, we further propose a linear insertion operator. Extensive experiments on real datasets validate the efficiency and scalability of our insertion operator. Particularly, the insertion operator can be accelerated by 1.5 to 998.1 times on urban-scale datasets.

Acknowledgment

We are grateful to anonymous reviewers for their constructive comments. This work is partially supported by National Science Foundation of China (NSFC) under Grant No. 61822201, U1811463, 71531001, the Science and Technology Major Project of Beijing under Grant No. Z171100005117001, and Didi Gaia Collaborative Research Funds for Young Scholars. Yongxin Tong is the corresponding author in this paper.

References

[1] Y. Tong, L. Chen, and C. Shahabi, "Spatial crowdsourcing: Challenges, techniques, and applications," *PVLDB*, vol. 10, no. 12, pp. 1988–1991, 2017.

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

[3] S. Ma, Y. Zheng, and O. Wolfson, "Real-time city-scale taxi ridesharing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1782–1795, 2015.

[4] Y. Huang, F. Bastani, R. Jin, and X. S. Wang, "Large scale real-time ridesharing with service guarantee on road networks," *PVLDB*, vol. 7, no. 14, pp. 2017–2028, 2014.

[5] P. Cheng, H. Xin, and L. Chen, "Utility-aware ridesharing on road networks," in *SIGMOD*, 2017, pp. 1197–1210.

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

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

[8] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu, "A unified approach to route planning for shared mobility," *PVLDB*, vol. 11, no. 11, pp. 1633–1646, 2018.

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

[10] M. Grötschel, S. O. Krumke, and J. Rambau, *Online Optimization of Large Scale Systems*, 2001.

[11] J. J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. M. Wilson, "A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows," *Transportation Research Part B*, vol. 20, no. 3, pp. 243–257, 1986.

[12] I. Ioachim, J. Desrosiers, Y. Dumas, M. M. Solomon, and D. Villeneuve, "A request clustering algorithm for door-to-door handicapped transportation," *Transportation Science*, vol. 29, no. 1, pp. 63–78, 1995.

[13] M. W. P. Savelsbergh and M. Sol, "The general pickup and delivery problem," *Transportation Science*, vol. 29, no. 1, pp. 17–29, 1995.

[14] J. F. Cordeau and G. Laporte, "The dial-a-ride problem: models and algorithms," *Annals of Operations Research*, vol. 153, no. 1, pp. 29–46, 2007.

[15] P. Toth and D. Vigo, *The vehicle routing problem*, 2002.

[16] Q. Tao, Y. Zeng, Z. Zhou, Y. Tong, L. Chen, and K. Xu, "Multi-worker-aware task planning in real-time spatial crowdsourcing," in *DASFAA*, 2018, pp. 301–317.

[17] P. M. Fenwick, "A new data structure for cumulative frequency tables," *Software: Practice and Experience*, vol. 24, no. 3, pp. 327–336, 1994.

[18] H. N. Psaraftis, *A Dynamic Programming Solution to the Single Vehicle Many-to-Many Immediate Request Dial-a-Ride Problem*, 1980, vol. 14, no. 2.

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

[20] B. Hunsaker and M. Savelsbergh, "Efficient feasibility testing for dial-a-ride problems," *Operations Research Letters*, vol. 30, no. 3, pp. 169–173, 2002.

[21] S. Yeung, E. Miller, and S. Madria, "A flexible real-time ridesharing system considering current road conditions," in *MDM*, vol. 1, 2016, pp. 186–191.

[22] Y. Tong, J. She, B. Ding, L. Wang, and L. Chen, "Online mobile micro-task allocation in spatial crowdsourcing," in *ICDE*, 2016, pp. 49–60.

[23] T. Song, Y. Tong, L. Wang, J. She, B. Yao, L. Chen, and K. Xu, "Trichromatic online matching in real-time spatial crowdsourcing," in *ICDE*, 2017, pp. 1009–1020.

[24] Y. Tong, L. Wang, Z. Zhou, B. Ding, L. Chen, J. Ye, and K. Xu, "Flexible online task assignment in real-time spatial data," *PVLDB*, vol. 10, no. 11, pp. 1334–1345, 2017.

[25] Y. Tong, L. Wang, Z. Zhou, L. Chen, B. Du, and J. Ye, "Dynamic pricing in spatial crowdsourcing: A matching-based approach," in *SIGMOD*, 2018, pp. 773–788.

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

[27] M. Diana and M. M. Dessouky, "A new regret insertion heuristic for solving large-scale dial-a-ride problems with time windows," *Transportation Research Part B*, vol. 38, no. 6, pp. 539–557, 2004.

[28] L. Coslovichaba, "A two-phase insertion technique of unexpected customers for a dynamic dial-a-ride problem," *European Journal of Operational Research*, vol. 175, no. 3, pp. 1605–1615, 2006.

[29] S. O. Krumke, W. de Paepe, D. Poensgen, M. Lipmann, A. Marchetti-Spaccamela, and L. Stougie, "On minimizing the maximum flow time in the online dial-a-ride problem," in *WAOA*, 2005, pp. 258–269.

[30] H. N. Psaraftis, *An Exact Algorithm for the Single Vehicle Many-to-Many Dial-A-Ride Problem with Time Windows*, 1983.

[31] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars, *Computational geometry: algorithms and applications, 3rd Edition*, 2008.

[32] TLC Tric Record Data. [Online]. Available: http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

[33] Geofabrik. [Online]. Available: https://http://www.geofabrik.de/

[34] Y. Tong, Y. Chen, Z. Zhou, L. Chen, J. Wang, Q. Yang, J. Ye, and W. Lv, "The simpler the better: A unified approach to predicting original taxi demands based on large-scale online platforms," in *SIGKDD*, 2017, pp. 1653–1662.

[35] Cainiao. [Online]. Available: https://en.wikipedia.org/wiki/China_Smart_Logistic_Network

[36] Tianchi. [Online]. Available: https://tianchi.aliyun.com/competition/introduction.htm?spm=5176.11409106.5678.1.738e63e3jmlVve&raceId=231581

[37] Y. Tong, J. She, B. Ding, L. Chen, T. Wo, and K. Xu, "Online minimum matching in real-time spatial data: Experiments and analysis," *PVLDB*, vol. 9, no. 12, pp. 1053–1064, 2016.

[38] Y. Zeng, Y. Tong, L. Chen, and Z. Zhou, "Latency-oriented task completion via spatial crowdsourcing," in *ICDE*, 2018, pp. 317–328.