This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3027200, IEEE Transactions on Knowledge and Data Engineering

An Efficient Insertion Operator in Dynamic Ridesharing Services

Yi Xu,Yongxin Tong, *Member, IEEE,* Yexuan Shi, Qian Tao, Ke Xu, and Wei Li

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/sum 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.

Index Terms—Insertion Operator, Dynamic Ridesharing, Dynamic Programming.

1 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. 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 [1], [2], [3], [4], [5]. It has been proved that there is no polynomial-time algorithm with a constant competitive ratio to solve the problem [5]. Hence many real-world ridesharing platforms, such as Didi Chuxing and Uber, rely on heuristic algorithms [1], [2], [5].

Insertion, or an "*insertion operator*", is widely adapted in various heuristic solutions to dynamic ridesharing [1], [2], [5], [6], [7], [8], [9] and is recognized as a core operator in these solutions [10], [11], [12]. 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/sum 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. Though a linear-time insertion method that optimizes the objective from the workers' perspective has been proposed [5], it cannot be adapted for the optimization objective from the requests' perspective as the insertion algorithm in [5] 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. Moreover, we harness efficient index structures, such as fenwick tree [13], 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 complexity 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.

A preliminary version of this work is in [14]. In this paper, we make the following new contributions: (1) We extend our partition-based framework to a new objective: total waiting time of requests (*i.e.,* sum flow time). (2) We

- *Y. Xu, Y. Tong, Y. Shi, Q. Tao, K. Xu and W. Li are with the State Key Laboratory of Software Development Environment and Advanced Innovation Center for Big Data and Brain Computing, School of Computer Science and Engineering, Beihang University, PR China. E-mail: {xuy, yxtong, skyxuan, qiantao, kexu}@buaa.edu.cn, liwei@nlsde.buaa.edu.cn.*
- *Yongxin Tong is the corresponding author in this paper.*

apply the segment-based optimization on the new objective. (3) We conduct new evaluations on real-world datasets.

In the rest of this paper, we define the insertion operator in Sec. 2 and review existing solutions in Sec. 3. For the two objectives which have max operator, we propose a partition-based framework in Sec. 4 and design a series of optimization techniques to reduce the time complexity in Sec. 5. In Sec. 6, we extend the framework and optimization techniques to a new objective: *sum flow time*. Finally we present the evaluations in Sec. 7 and conclude in Sec. 8.

2 Problem Statement

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

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.*

We 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 these 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.*

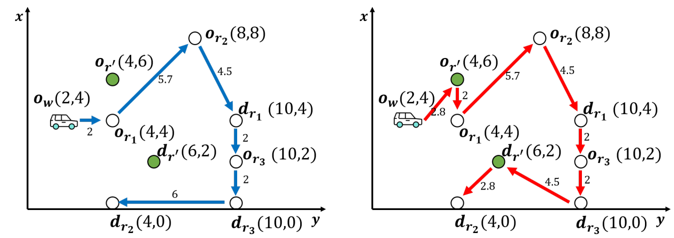
(1) ***Maximum flow time*** *of all the requests [6], [7], [15], [16], i.e. $\max_{r \in R^+}\{flw(r)\}$.*

(2) ***Total travel time*** *of the worker [1], [2], [5], [9], or equivalently,* ***the delivery time of the last request****, i.e. $\max_{r \in R^+}\{delv(r)\}$.*

(3) ***Sum flow time*** *of all the requests [17], [18], i.e. $\sum_{r \in R^+}\{flw(r)\}$.*

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.



(a) Route before insertion (b) Route before insertion

Fig. 1: An example of insertion.

TABLE 1: 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

- For convenience, we rewrite the three objective functions into a unified form as

$$\text{OBJ}(S_{R^+}) = \underset{r \in R^+}{\texttt{OP}} \{flw(r) + \alpha \cdot t_r\}, \tag{1}$$

where α is either 1 or 0. Note that

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

The following example illustrates the insertion operator.

Example 1. *Suppose that on a ridesharing platform a driver w serves three requests r_1-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 1. 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 $\max\{17, 28.3, 21, 23.5\} = 28.3$*; the total travel time of the route is $2 + 2.8 + 2 + 5.7 + 4.5 + 2 + 2 + 4.5 + 2.8 = 28.3$ and the sum flow time of the route is $17 + 28.3 + 21 + 23.5 = 89.8$.*

3 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 [19], [20], [21], [22], [23]. The first research paper dates back to the pickup and delivery problem (*a.k.a.* dial-a-ride problem) proposed in 1975 [24], 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 [1], [2], [5], [6],

TABLE 2: Time complexity for *insertion* in existing works.

Method and Reference	Objective	Time
adaptive insertion [6]	max flow time	$O(n^3)$
large-scale insertion [7]	max flow time	$O(n^3)$
sequential insertion [8]	max flow time	$O(n^3)$
	sum flow time	$O(n^3)$
regret insertion [17]	sum flow time	$O(n^3)$
two-phrase insertion [18]	sum flow time	$O(n^3)$
clustering insertion [9]	total travel time	$O(n^3)$
tshare [1]	total travel time	$O(n^3)$
kinetic [2], [3], [4]	total travel time	$O(n^2)$
pruneGreedyDP [5]	total travel time	$O(n)$
our approach in this paper	max flow time	$O(n)$
	sum flow time	
	total travel time	

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^+};

[7], [8], [9], [17], [18] and is regarded as a basic operator in ridesharing [10], [11]. Table 2 lists some of the most representative solutions to ridesharing based on insertion under different optimization objectives.

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 of different optimization objectives below.

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, Jaw *et al.* [8] propose to sequentially insert requests into the current route, which can handle a few thousands (around 3000) of requests. This insertion procedure is widely used by follow-up papers [6], [7], [25]. Hame *et al.* [6] also utilize insertion to adaptively solve the problem. For larger-scale datasets, Krumke *et al.* [7], [25] design a batch based framework where insertion can be directly used. The insertion to minimize the maximum flow time takes $O(n^3)$ time [6], [7], [8].

Sum flow time models the average waiting time of the requests. Jaw *et al.* [8] first devise a cubic-time insertion operator for this objective. This insertion method has been adopted by many other solutions [17], [18] to minimize the sum flow time. For instance, Diana *et al.* [17] propose a new regret insertion technique, which swaps two requests from two different routes by re-inserting each request into the other route. Coslovichaba [18] proposes a two-phase framework for real-time dia-a-ride.

Total travel time indicates the preference of workers [26], *i.e.*, a worker usually wants to serve all requests in less time.

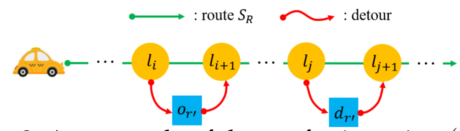


Fig. 2: An example of detour for insertion (i, j).

To minimize the total travel time in ridesharing, Iochim *et al.* [9] cluster the nearest requests first and then construct the route for each worker by repeated insertion. They use the insertion procedure of [8] in $O(n^3)$ time and insert requests into different routes parallel. Zheng *et al.* [1] design a general framework that repeatedly executes an $O(n^3)$ insertion. Huang *et al.* [2] combines insertion and kinetic tree such that the time complexity of insertion is reduced to $O(n^2)$. Kinetic tree is widely used to minimize the total travel time of ridesharing [3], [4]. Tong *et al.* [5] further accelerate the insertion operator to minimize the total travel time to linear time, which has been applied in [27].

In summary, insertion is the cornerstone of many existing solutions to ridesharing. Although insertion with linear time has been proposed for one special optimization objective, 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.

4 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 partitiobn 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. 4.1, based on which we devise an insertion operator of $O(n^2)$ time complexity using dynamic programming in Sec. 4.2. This section focuses on maximum flow time and total travel time since these two objectives are the cases where $\texttt{OP} = \texttt{max}$ (see Eq.(2)). We discuss extensions to sum flow time in Sec. 6.1.

4.1 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:

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

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 $(i + 1)$-th location) and the detour of inserting $d_{r'}$ (the increased travel time from the j-th location and $(j + 1)$-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).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3027200, IEEE Transactions on Knowledge and Data Engineering

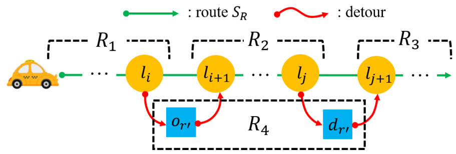


Fig. 3: An example of request partition ($i < j$).

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 deadline 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;

(1) 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 $d_{r'}$.

(2) 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 $o_{r'}$.

(3) 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 $d_{r'}$.

(4) R_4 only contains 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. 4.2.1). 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.

4.2 Naive Dynamic Programming Based Insertion

This subsection introduces an $O(n^2)$ insertion operator based on the partition framework in Sec. 4.1. 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 3 summarizes the major notations.

4.2.1 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 his capacity and 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 $slk(\cdot)$.

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 $0 \le k \le n$, $pck(k)$ can be pre-calculated in $O(n)$. With $pck(k)$ we can check the capacity constraint in $O(1)$ through Lemma 1.

TABLE 3: 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

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

The proof of Lemma 1 can be found in [14].

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,

$$slk(k) = \min\{slk(k+1), ddl(k+1) - arr(k+1)\}, \tag{4}$$

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

$$ddl(k) = \begin{cases} e_r - dis(o_r, d_r), & l_k \text{ is an origin} \\ e_r, & l_k \text{ is a destination.} \end{cases} \tag{5}$$

The value of $slk(k)$ for all $0 \le k \le n$ 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.

(1) Check whether any deadline constraint of all the existing requests is violated by inserting $o_{r'}$ after l_i, *i.e.*, whether $det(i, o_{r'}) \le slk(i)$;

(2) 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 $i < j$;

(3) 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 $i < j$.

4.2.2 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)* $i = j$.

***(i)* 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.

(1) 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

$$mf_1 = mobj(0, i). \tag{6}$$

(2) 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3027200, IEEE Transactions on Knowledge and Data Engineering

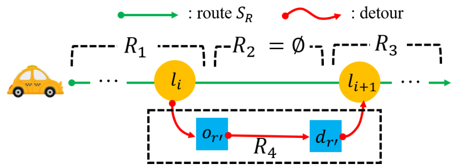


Fig. 4: An example of requests partition ($i = j$).

TABLE 4: 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

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

$$mf_2 = det(i, o_{r'}) + mobj(i+1, j). \tag{7}$$

(3) 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 calculated as

$$mf_3 = det(i, o_{r'}) + det(j, d_{r'}) + mobj(j+1, n). \tag{8}$$

(4) 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

$$mf_4 = arr(j) + det(i, o_{r'}) + dis(l_j, d_{r'}) + (\alpha - 1)t_{r'}. \tag{9}$$

***(ii)* 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: *1)* R_2 contains no requests when $i = j$; and *2)* 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.

(1) Calculating mf_1: mf_1 is still $mobj(0, i)$ since the requests in R_1 are not influenced by detour.

(2) Calculating mf_2: mf_2 is 0 because R_2 contains no requests when $i = j$.

(3) 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 $det(i, r') + mobj(i+1, n)$.

(4) 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

$$mf_4 = arr(i) + dis(l_i, o_{r'}) + dis(o_{r'}, d_{r'}) + (\alpha - 1)t_{r'}. \tag{10}$$

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 4. Take $i = 1, j = 3$ as an example. 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 calculate $mobj(1, 4)$, $mobj(1, 5)$ and $mobj(1, 6)$.*

Then we 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 $mf_2 = det(1, o_{r'}) + mobj(2, 5) = 19$ (Eq.(7)). As for the requests in R_3, we have $det(1, o_{r'}) = 0.8$ and $det(5, d_{r'}) = 1.3$. Thus, $mf_3 = det(1, o_{r'}) + det(5, d_{r'}) + mobj(6, 6) = 26.3$. To obtain the maximum flow time of requests in R_4, we first get $arr(5) = 18.2$, $det(1, o_{r'}) = 0.8$ and $dis(l_5, d_{r'}) = 4.5$. Substituting these results into Eq.(9), 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 $\max\{0, 19, 26.3, 23.5\} = 26.3$.

TABLE 5: 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(×)

Algorithm 3: Naive DP Algorithm

```
1  S_{R+} ← S_R, O* ← ∞, i* ← none, j* ← none;
2  Pre-calculate pck(·), slk(·), mobj(·,·);
3  for i ← 0 to n do
4      for j ← i to n do
5          if capacity constraint is violated then break ;
6          if deadline constraint is violated then continue ;
7          mf_1, mf_2, mf_3, mf_4 ← obtain by Eq.(6)-(10);
8          O ← max{mf_1, mf_2, mf_3, mf_4};
9          if O < O* then
10             O* ← O, i* ← i, j* ← j;
```

4.2.3 Algorithm Details

Alg. 3 illustrates the naive DP based insertion algorithm. In line 2, we pre-calculate $pck(\cdot), slk(\cdot), mobj(\cdot, \cdot)$ as in Sec. 4.2.1 and Sec. 4.2.2. While enumerating the pairs (i, j) for insertion in lines 3-4, we first check the capacity constraint in line 5. If it is violated, we directly break the enumeration of j by 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.(6)-Eq.(10) 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.

Example 3. *Back to Example 1. Table 5 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 4. Take $i = 1$ as an example. For each j from 1 to 6, we first check the capacity and deadline constraints. The insertions $(1, 1)$ to $(1, 5)$ satisfy the constraints. We further calculate the maximum flow time as 31.3, 31.3, 31.5, 31.5 and 26.3 respectively. So we know insertion $(1, 5)$ leads to the minimum maximum flow time of requests.*

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 $O(n^2)$. Thus, the naive DP based insertion has a time complexity of $O(n^2)$ and a space complexity of $O(n^2)$.

5 A Segment-based DP Algorithm

In this section, we push the limit of the time complexity of the generic insertion operator from $O(n^2)$ 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. 5.1, and then present key observations on the capacity and the deadline constraints in Sec. 5.2. Accordingly, we introduce the basic idea of the segment-based DP algorithm in Sec. 5.3, and describe the detailed algorithm in Sec. 5.4. Its extension to minimize sum flow time will be discussed in Sec. 6.2.

5.1 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.

(1) 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 as $mobj(j+1, n)$ 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

$$det(i, o_{r'}) + \max\{mobj(i+1, n), det(j, d_{r'}) + mobj(j+1, n)\}. \tag{11}$$

(2) Calculating com_2: Since $mf_1 = mobj(0, i)$, we have $com_2 = \max\{mobj(0, i), com_1\}$. Based on Eq.(11), $mobj(i+1, n)$ is no larger than com_1. Thus we can safely combine $mobj(i+1, n)$ into com_2 as:

$$\begin{aligned} com_2 &= \max\{mobj(0, i), mobj(i+1, n), com_1\} \\ &= \max\{mobj(0, n), com_1\}. \end{aligned} \tag{12}$$

(3) Calculating Objectives: To calculate 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.

$$\begin{aligned} par(j) = \max\{&det(j, d_{r'}) + mobj(j+1, n), \\ &arr(j) + dis(l_j, d_{r'}) + (\alpha-1)t_{r'}\}. \end{aligned} \tag{13}$$

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

$$\max\Big\{mobj(0, n), det(i, o_{r'}) + \max\{mobj(i+1, n), par(j)\}\Big\}. \tag{14}$$

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 (enumerated), 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.

5.2 Observations on 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 $brk(i+1)$.*

Observation on deadline constraint. According to the deadline constraints in Sec. 4.2.1, we have the following observation, as illustrated in Lemma 3.

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

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

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. $thr(j) \geq det(i, o_{r'})$.

The proof of Lemma 2-3 can be found in [14].

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 next subsections, as we enumerate i, we aim to calculate the minimum objective from such j more efficiently, *i.e.*,

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

5.3 Segment-based Optimization

Basic Idea: If we enumerate i, by utilizing data structure like segment tree [28], 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.*, $thr(j) \geq det(i, o_{r'})$), 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 each leaf node in the tree. Thus, we can efficiently query the minimum value of $par(j)$ among previously filtered

Algorithm 4: Segment-based DP Algorithm

```
1  S_{R+} ← S_R, O* ← ∞, i* ← none, j* ← none;
2  Pre-calculate pck(·), slk(·), thr(·), mobj(·, n);
3  Construct a segment tree ST;
4  for i ← 0 to n do
5      Handle the case when i = j;
6  for i ← n − 1 to 0 do
7      Update leaf 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) ≤ c_w − c_{r'} and det(i, o_{r'}) ≤ slk(i) then
11         Query the minimum par(j) from segment
             [det(i, o_{r'}), ∞) in ST;
12         O ← calculate objective according to Eq.(16);
13         if O < O* then
14             O* ← O, i* ← i, j* ← j;
```

positions. As a result, we can efficiently calculate Eq.(15) for a fixed i. Specifically, the terms in Eq.(14) like $mobj(0,n)$, $det(i,o_{r'})$, $mobj(i+1,n)$ are constant for a fixed i. Substituting Eq.(14) into Eq.(15), we have:

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

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.

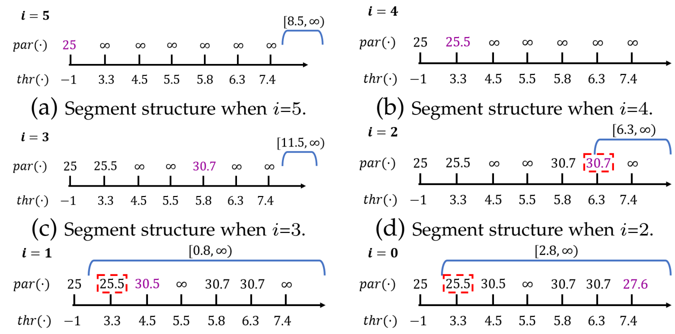
5.4 Algorithm Details

Alg. 4 illustrates the segment-based DP insertion algorithm. In line 2, we pre-calculate $pck(\cdot)$, $slk(\cdot)$, $thr(\cdot)$, $mobj(\cdot,n)$ as in Sec. 4.2. 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 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 according to Eq.(16). In lines 13-14, we update O^*, i^* and j^* according to the current objective O.

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). With the dynamic index structures, we can only maintain the feasible j, which leads to a $O(1)$ maintain time.

TABLE 6: 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



(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.

Example 4. *Back to the settings in Example 1. We aim to find the minimum maximum flow time of requests. Table 6 summarizes the values after pre-calculation. We have obtained the values of $mobj(\cdot,6)$ in Table 4. The values of $thr(\cdot)$ and $par(\cdot)$ are calculated by their definitions in Sec. 5.2 and Sec. 5.3, respectively.*

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 $i+1$ 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 objective 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 objective 26.3. Similarly the case $i=0$ leads to the objective 28.3. Finally we have the minimum objective 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**. Lines 4-5 take $O(n)$ time. In the iterations 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, and other lines take $O(1)$ 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 [13] (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 $O(n)$.

6 Extension to Sum Flow Time

In this section, we extend the partition-based framework and segment-based optimization technique to the objective of sum flow time. We can use the same way to check the capacity and deadline constraints as in Sec. 4 and Sec. 5 because the constraints are the same for each objective. Therefore, we mainly focus on efficiently calculating the objective value, as will be explained in the following subsections.

6.1 Extension of Partition-based Framework

Within the partition framework, the objective of minimizing the sum flow time, *i.e.*, Eq.(1), is rewritten as

$$\mathrm{OBJ}(S_{R^+}) = sf_1 + sf_2 + sf_3 + sf_4 \tag{17}$$

where

$$sf_1 = \sum_{r\in R_1} flw(r), \qquad sf_2 = \sum_{r\in R_2} flw(r),$$
$$sf_3 = \sum_{r\in R_3} flw(r), \qquad sf_4 = \sum_{r\in R_4} flw(r).$$

Now we show how to calculate sf_1, sf_2, sf_3 and sf_4 in $O(1)$ time when enumerating i and j. Similar to the notation $mobj(i,j)$ in Sec. 4.2.2, we use $sobj(i,j)$ to denote the sum of the flow time ($flw(r)$) of the requests (r), whose destinations are between the i-th location and j-th location. Further denote $num(i,j)$ as the number of such requests, whose destinations are between the i-th location and j-th location. It takes $O(n^2)$ time to pre-calculate $sobj(i,j)$ and $num(i,j)$ by enumerating i from 0 to n and j from i to n.

Next, we show how to calculate sf_1, sf_2, sf_3 and sf_4 in two cases: $i<j$ and $i=j$.

Calculations in Case of $i<j$.

(1) Calculating sf_1: All the requests in R_1 (whose destination is before the i-th location) are not influenced by the detour. Thus, sf_1 can be calculated as

$$sf_1 = sobj(0,i). \tag{18}$$

(2) Calculating sf_2: 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, the flow time ($flw(r)$) of each request in R_2 would increase by $det(i,o_{r'})$. Thus sf_2 can be calculated as

$$sf_2 = num(i+1,j)\times det(i,o_{r'}) + sobj(i+1,j). \tag{19}$$

(3) Calculating sf_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, the flow time ($flw(r)$) of each request in R_3 would increase by $det(i,o_{r'}) + det(j,d_{r'})$. Thus sf_3 can be calculated as

$$sf_3 = num(j+1,n)\times[det(i,o_{r'})+det(j,d_{r'})] + sobj(j+1,n). \tag{20}$$

(4) Calculating sf_4: 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, according to the definition of flow time ($flw(r)$), we have

$$sf_4 = arr(j) + det(i,o_{r'}) + dis(l_j,d_{r'}) - t_{r'}. \tag{21}$$

Calculations in Case of $i=j$.

(1) Calculating sf_1: sf_1 is still $sobj(0,i)$ since the requests in R_1 are not influenced by detour.

(2) Calculating sf_2: sf_2 is 0 because R_2 contains no requests when $i=j$.

(3) Calculating sf_3: Denote $det(i,r')$ as the detour when $i=j$. Then $det(i,r')$ can be calculated as

$$dis(l_i,o_{r'}) + dis(o_{r'},d_{r'}) + dis(d_{r'},l_{i+1}) - dis(l_i,l_{i+1}).$$

Since all the requests in R_3 are influenced by this detour, their flow time would increase by $det(i,r')$. Thus mf_3 can be calculated as $num(i+1,n)\times det(i,r') + sobj(i+1,n)$.

(4) Calculating sf_4: For sf_4, the worker 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 sf_4 can be calculated as

$$sf_4 = arr(i) + dis(l_i,o_{r'}) + dis(o_{r'},d_{r'}) - t_{r'}. \tag{22}$$

Algorithm Details. We show how to extend the naive DP algorithm to the objective of minimizing the *sum flow time*. Back to Alg. 3, we need to pre-calculate an extra array $num(i,j)$ in line 2. In the iterations of lines 3-10, we will calculate sf_1, sf_2, sf_3, sf_4 by Eq.(18)-Eq.(22) in line 7 and then calculate the objective by summing sf_1 to sf_4 in line 8. All the other lines remain the same as in Alg. 3.

Example 5. *Back to Example 1. Suppose we want to calculate the sum flow time of insertion $(1,5)$. Since $sobj(\cdot,\cdot)$ and $num(\cdot,\cdot)$ can be simply obtained as shown in Sec. 6.1, we omit their calculations and focus on calculating sf_1, sf_2, sf_3 and sf_4. First the sum flow time of requests in R_1 is $sf_1 = sobj(0,1) = 0$. Since $det(1,o_{r'}) = 0.8$, the sum flow time of requests in R_2 is $sf_2 = num(2,5)\times det(i,o_{r'}) + sobj(2,5) = 34$ (Eq.(19)). As for the requests in R_3, we have $det(1,o_{r'}) = 0.8$ and $det(5,d_{r'}) = 1.3$. Thus, $sf_3 = num(6,6)\times[det(i,o_{r'}) + det(j,d_{r'})] + sobj(6,6) = 0.8+1.3+24.2 = 26.3$. To obtain the sum 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.(21), we have that the flow time of the new request (R_4) is $sf_4 = arr(5) + det(1,o_{r'}) + dis(l_5,d_{r'}) = 23.5$. Finally the sum flow time for insertion $(1,5)$ is $0+34+26.3+23.5 = 83.8$.*

Complexity Analysis. The time complexity of the extension is still $O(n^2)$, since it takes $O(n^2)$ time to pre-calculate the auxiliary array $num(\cdot,\cdot)$ and the calculation of the objective value is $O(1)$.

6.2 Extension of Segment-based DP algorithm

To extend the segment-based DP algorithm, we also use the same way to check the constraints and only focus on the objective calculation in the following.

Basic Idea. Here we only focus on the case of $i<j$ because the case of $i=j$ can be done in $O(n)$ naturally. As in Sec. 5.3, we only need a column ($j=n$) of the arrays $sobj$ and num to compute the objective. Specifically, we can first compute $sf_1+sf_2+sf_3$ and then add it to sf_4. This is because sf_4

does not relate to either $sobj$ or num. An optimized way to calculate $sf_1 + sf_2 + sf_3$ is as follows.

$$\begin{aligned}
&sf_1 + sf_2 + sf_3 \\
&= [sobj(0,i) + sobj(i+1,j) + sobj(j+1,n)]+ \\
&\quad [num(i+1,j) + num(j+1,n)] \times det(i, o_{r'})+ \\
&\quad num(j+1,n) \times det(j, d_{r'}) \\
&= sobj(0,n) + num(i+1,n) \times det(i, o_{r'})+ \\
&\quad num(j+1,n) \times det(j, d_{r'}).
\end{aligned} \tag{23}$$

Summing Eq.(23) and Eq.(21), we can rewrite Eq.(17) as:

$$\begin{aligned}
&\mathrm{OBJ}(S_{R^+}) = sf_1 + sf_2 + sf_3 + sf_4 \\
&= sobj(0,n) + \big(num(i+1,n)+1\big) \cdot det(i, o_{r'}) - t_{r'} + spar(j),
\end{aligned} \tag{24}$$

where $spar(j)$ is the sum of all the terms related to j, *i.e.*, $spar(j) = num(j+1,n) \times det(j, d_{r'}) + arr(j) + dis(l_j, d_{r'})$.

According to Eq.(24) and the observations on the constraints in Sec. 5.2, we can finally rewrite Eq.(16) as:

$$\begin{aligned}
sobj(0,n) + &\big(num(i+1,n)+1\big) \times det(i, o_{r'}) - t_{r'} \\
&+ \min_{i<j<brk(i), thr(j) \geq det(i, o_{r'})} \{spar(j)\}.
\end{aligned} \tag{25}$$

In Eq.(25), only the last term is related to j. Hence the sum of other terms is constant for a fixed j. We can also apply segment tree or fenwick tree to efficiently query the result of the last term as in Sec. 5.3.

Algorithm Details. In the extended version of Alg. 4, we do not need to calculate $mobj(\cdot, n)$ in line 2 any more. Instead, we calculate $num(\cdot, n)$ and $sobj(\cdot, n)$. In line 7 and line 11, we maintain the value of $spar(j)$ rather than $par(j)$. In line 12, we calculate the objective by Eq.(25). All the other lines remain the same as in Alg. 4.

Complexity Analysis. The time complexity of the extended version is the same as the original one, since each line takes the same time. For instance, the time complexity is $O(n \log n)$ by segment tree and $O(n)$ by fenwick tree.

7 Experimental Study

This section presents the evaluation of our algorithms.

7.1 Experimental Setup

Datasets. We experiment with two real datasets (Table 7).

The first dataset [5] (denoted by ***Taxi***) is the trip records of taxis in New York City, which has over $517k$ requests in a single day (2nd row in Table 7). We use the same methods as in [5], [14] to process these requests. Since there is no information about the workers, we uniformly generate the workers' locations on the road network in ***Taxi***. We also generate the capacities of the workers by a Gaussian distribution whose mean varies from 3 to 20 (2nd row of Table 8). Considering that short trips dominate in the requests, 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 8). To test the algorithms with different amounts of requests, we extract the first 20k to 100k requests for evaluation (3rd row of Table 8). To test the scalability of the algorithms, we extract the first 100k to 500k requests for evaluation (5th row of Table 8). Note the number of requests here denotes the total number of requests instead of the number of requests assigned to one worker (*i.e.*, n). The default settings are marked in bold.

TABLE 7: Statistics of datasets.

Dataset	Space	#(Requests)	#(Vertices)	#(Edges)
Taxi	Road network	517,100	807,795	2,100,632
Parcel	Euclidean space	345,849	12,487	–

TABLE 8: Parameter settings.

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

The second dataset [14] (denoted by ***Parcel***) comes from Cainiao [29] , a well-known parcel delivery platform in China. 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 7). In ***Parcel***, the distance between two locations is their Euclidean distance. We pre-process ***Parcel*** in a similar way to ***Taxi*** and the parameter settings are shown in Table 8. 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 ***Parcel***.

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

(1) BF (Alg. 1) is an $O(n^3)$-time insertion operator.

(2) NDP (Alg. 3) is an $O(n^2)$-time insertion operator by naive dynamic programming (DP).

(3) ST (Alg. 4 with segment tree implementation) is an $O(n \log n)$-time insertion operator by DP and segment tree.

(4) FT (Alg. 4 with fenwick tree implementation) is an $O(n)$-time insertion operator by DP and fenwick tree.

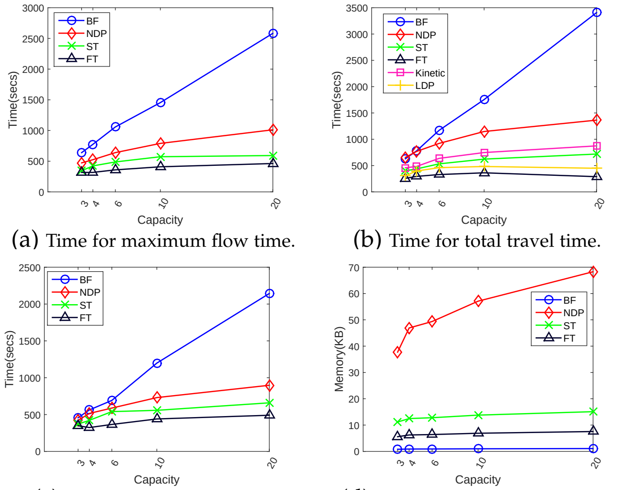
(5) Kinetic [2] is an existing $O(n^2)$-time insertion operator for minimizing the total travel time.

(6) LDP [5] is the state-of-the-art $O(n)$-time insertion operator for minimizing the total travel time.

Note that **LDP** and **Kinetic** are only applicable to minimizing the total travel time. Hence we exclude these two algorithms in the experiments of other objectives.

Implementation. The experiments are conducted on a server with 40 Intel(R) Xeon(R) E5 2.30GHz processors with 128GB memory. All of the algorithms are implemented in GNU C++. Each experiment is repeated 30 times and we show the average results.

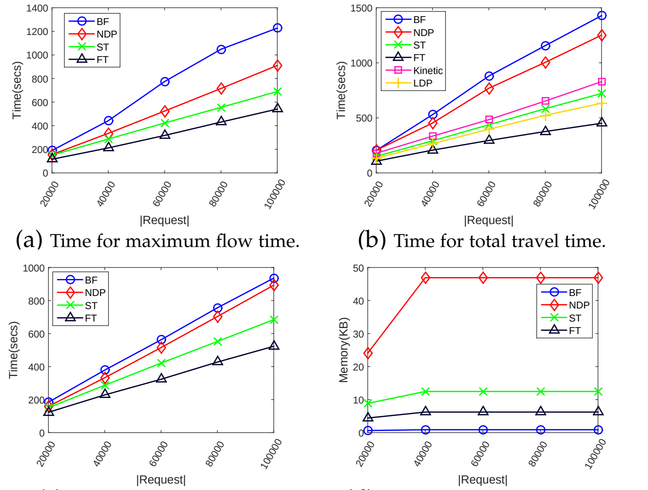
Metrics. We integrate the above insertion algorithms into a widely used route planning solution to dynamic ridesharing [1], [2], [5]. Upon arrival of a new request, the solution inserts a new request to all possible workers who can pick up the request using the insertion operator and greedily returns the best insertion locations and the corresponding worker. As previous works like [30], [31], we compare the memory and time cost 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* when using different insertion operators for ridesharing. As for the memory cost, we only consider the memory caused by insertion operators and exclude the common memory like spatial indices and road network.



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

(c) Time for sum flow time. (d) Memory for sum flow time.

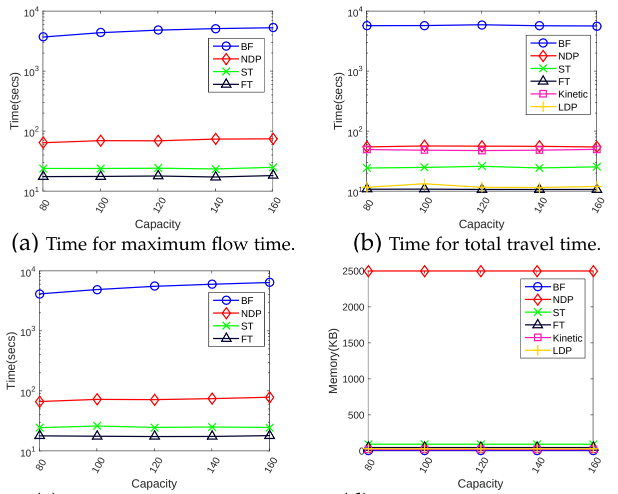
Fig. 6: Results of varying capacity of workers on ***Taxi***.



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

(c) Time for sum flow time. (d) Memory for sum flow time.

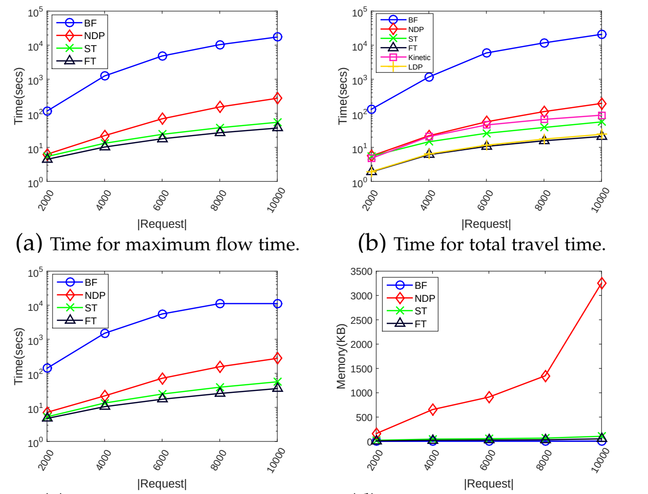
Fig. 8: Results of varying # of requests on ***Taxi***.



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

(c) Time for sum flow time. (d) Memory for sum flow time.

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



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

(c) Time for sum flow time. (d) Memory for sum flow time.

Fig. 9: Results of varying # of requests on ***Parcel***.

7.2 Experimental Results

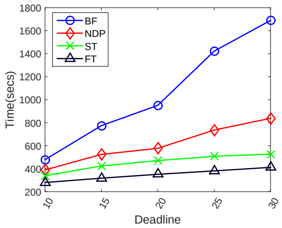
Due to the limit of space, we omit the figures of memory consumption when minimizing the maximum flow time and total travel time, which can be found in [14]. Note that the memory trend of all these three objectives are similar.

Impact of Capacity of Workers. Fig. 6 and Fig. 7 show the results of varying the capacity of workers on ***Taxi*** and ***Parcel***, respectively. FT has the shortest running time in all of the three objectives, which is up to 6.4 and 301.8 times faster than the others on ***Taxi*** and ***Parcel***, respectively. Specifically, when minimizing the total travel time, FT is sometimes 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 ***Parcel***, the time and memory 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.

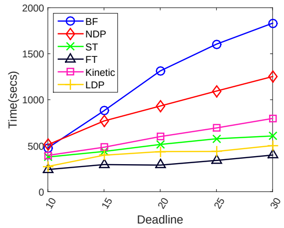
Impact of Number of Requests. Fig. 8 and Fig. 9 show the results of varying the number of requests on ***Taxi*** and ***Parcel***, respectively. FT still outperforms the other algorithms in terms of the average running time when minimizing the maximum/sum flow time, *i.e.,* 2.2 and 998.1 times faster than BF on ***Taxi*** and ***Parcel***, respectively. When minimizing the total travel time, FT is faster than LDP on ***Taxi*** and is as fast as LDP on ***Parcel*** 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 ***Parcel***. 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. 10 shows the results of varying the deadline on ***Taxi***. FT is again the fastest among all the algorithms, which is up to 4.6 times faster. With the increase of $e_r - t_r$, the time costs of all the algorithms

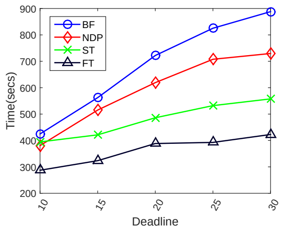
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3027200, IEEE Transactions on Knowledge and Data Engineering



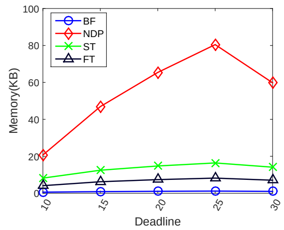
(a) Time for maximum flow time.



(b) Time for total travel time.

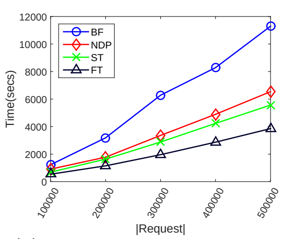


(c) Time for sum flow time.

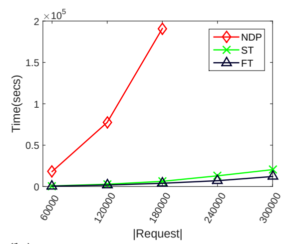


(d) Memory for sum flow time.

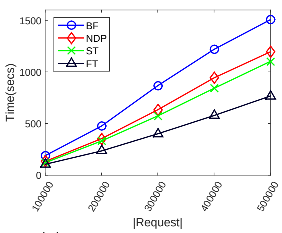
Fig. 10: Results of varying $e_r - t_r$ on ***Taxi***.



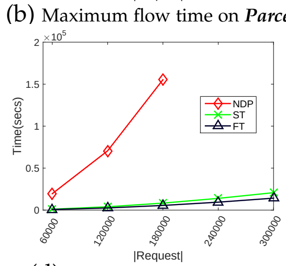
(a) Maximum flow time on ***Taxi***.



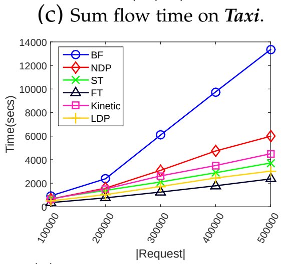
(b) Maximum flow time on ***Parcel***.



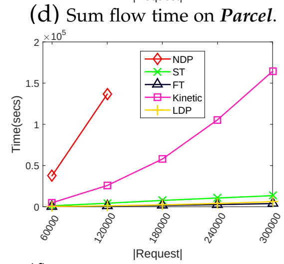
(c) Sum flow time on ***Taxi***.



(d) Sum flow time on ***Parcel***.



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



(f) Total travel time on ***Parcel***.

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

increase, while those of FT and LDP increase slower than BF, Kinetic, ST and NDP. This is because with a larger deadline, more 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. We also observe the memory costs of all algorithms decrease in the end. The reason is as follows. When $e_r - t_r$ is 25-30 minutes, each request has more feasible workers for insertion. Thus, the number of requests assigned to each worker is more balanced, which leads to less (peak) memory cost.

Scalability. Fig. 11 shows the experimental results of time cost on scalability. On ***Parcel***, BF and NDP fail to terminate in two days and hence we omit their results. Among all the three objectives, FT is always the most efficient, and ST is often the runner-up (less efficient than LDP when minimizing the total travel time). The results show that both ST and FT are fit for large-scale datasets.

Comparison between Datasets. Comparing the results on ***Taxi*** and ***Parcel***, we have the following observations.

- On both datasets FT outperforms the other algorithms in terms of running time, except for Fig. 9b where LDP runs as fast as FT.
- All the algorithms consume more space on ***Parcel*** (40-2500 KB) than on ***Taxi*** (10-140 KB). This may be because the requests in ***Parcel*** have a larger capacity. 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 ridesharing applications (more than 24 hours on ***Parcel***).
- 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 ***Parcel***.
- Our ST algorithm is up to 6.8 times faster than NDP on two datasets, while our FT algorithm is even faster, *i.e.*, up to 8.3 times faster than NDP.
- Our FT algorithm is the most efficient when minimizing the maximum/sum flow time. For the other objective, our algorithm FT runs faster than LDP, the state-of-the-art insertion operator to minimize the total travel time, in most of the experiments.
- The memory costs of ST and FT are only slightly larger (within 0.1 MB) than the memory usage of BF.

8 Conclusion

In this paper, we focus on the insertion operator, a widely used core operation in real-world dynamic ridesharing applications. Specifically, we study the efficient insertion operation for three practical objectives: minimizing the maximum/sum flow time of the requests and the total travel time of the workers. A straightforward implementation of the insertion operator takes $O(n^3)$ time to obtain the optimal insertion locations. To improve the efficiency, we propose a partition-based 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-time insertion operator for all the three objectives. 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.

References

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

[2] 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3027200, IEEE Transactions on Knowledge and Data Engineering

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] Y. Xu, Y. Tong, Y. Shi, Q. Tao, K. Xu, and W. Li, “An efficient insertion operator in dynamic ridesharing services,” in *ICDE*, 2019, pp. 1022–1033.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] K. Li, L. Chen, and S. Shang, “Towards alleviating traffic congestion: Optimal route planning for massive-scale trips,” in *IJCAI*, C. Bessiere, Ed., 2020, pp. 3400–3406.

[22] L. Chen, S. Shang, B. Yao, and J. Li, “Pay your trip for traffic congestion: Dynamic pricing in traffic-aware road networks,” in *AAAI*, 2020, pp. 582–589.

[23] L. Chen, S. Shang, C. Yang, and J. Li, “Spatial keyword search: a survey,” *GeoInformatica*, vol. 24, no. 1, pp. 85–106, 2020. [Online]. Available: https://doi.org/10.1007/s10707-019-00373-y

[24] N. H. Wilson, R. Weissberg, B. Higonnet, and J. Hauser, “Advanced dial-a-ride algorithms,” Tech. Rep., 1975.

[25] 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.

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

[27] Y. Zeng, Y. Tong, and L. Chen, “Last-mile delivery made practical: An efficient route planning framework with theoretical guarantees,” *PVLDB*, vol. 13, no. 3, pp. 320–333, 2019.

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

[29] Cainiao. [Online]. Available: https://www.cainiao.com

[30] 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.

[31] H. Luo, Z. Bao, F. Choudhury, and S. Culpepper, “Dynamic ridesharing in peak travel periods,” *IEEE Transactions on Knowledge and Data Engineering*, 2019.

Yi Xu is currently working toward the Ph.D. degree in the School of Computer Science and Engineering, Beihang University. His research interests include big spatio-temporal data analytics and mining, crowd intelligence, crowdsourcing and privacy preserving data analytics.

Yongxin Tong received the Ph.D. degree in computer science and engineering from the Hong Kong University of Science and Technology in 2014. He is currently a professor in the School of Computer Science and Engineering, Beihang University. His research interests include big spatio-temporal data analytics, crowdsourcing, crowd intelligence, federated learning, privacy preserving data analytics and uncertain data management. He is a member of the IEEE.

Yexuan Shi is currently working toward the Ph.D. degree in the School of Computer Science and Engineering, Beihang University. His major research interests include big spatio-temporal data analytics, crowdsourcing and privacy preserving data analytics.

Qian Tao is currently working toward the Ph.D. degree in the School of Computer Science and Engineering, Beihang University. His major research interests include big spatio-temporal data analytics, crowdsourcing and privacy preserving data analytics.

Ke Xu is a professor in the School of Computer Science and Engineering, Beihang University, China. He received his B.E, M.E. and Ph.D. degree from Beihang University in 1993, 1996, and 2000, respectively. His current research interests include phase transitions in NP-Complete problems, algorithm design, computational complexity, big spatio-temporal data analytics, crowdsourcing and crowd intelligence.

Wei Li is a professor in the School of Computer Science and Engineering, Beihang University, China. He is a Member of the Chinese Academy of Sciences and the Academia Europaea. He was a president of Beihang University from 2002 to 2009 and a director of the State Key Laboratory of Software Development Environment, China. He received his Ph.D. degree in Computer Science from the University of Edinburgh, and his B.S. degree in Mathematics from Peking University. His current research interests include mathematical logic, big data, artificial intelligence, smart city, crowdsourcing and crowd intelligence.