2018 IEEE 34th International Conference on Data Engineering

Price-and-Time-Aware Dynamic Ridesharing

Lu Chen[#,‡1], Qilu Zhong[#2], Xiaokui Xiao[§3], Yunjun Gao[#,†4], Pengfei Jin[#5], Christian S. Jensen[‡6]

[#]*College of Computer Science, Zhejiang University, Hangzhou, China*
[§]*School of Computing, National University of Singapore, Singapore*
[†]*The Key Lab of Big Data Intelligent Computing of Zhejiang Province, Zhejiang University, Hangzhou, China*
[‡]*Department of Computer Science, Aalborg University, Denmark*

{[1]luchen, [2]qlzhong, [4]gaoyj, [5]pfjin}@zju.edu.cn [3]xkxiao@nus.edu.sg [6]csj@cs.aau.dk

***Abstract*—Ridesharing refers to a transportation scenario where travellers with similar itineraries and time schedules share a vehicle for a trip and split the travel cost, which may include fuel, tolls, and parking fees. Ridesharing is popular among travellers because it can reduce their travel costs, and it also holds the potential to reduce travel time, congestion, air pollution, and overall fuel consumption. However, existing ridesharing systems often offer each traveller only one choice that aims to minimize system-wide vehicle travel distance or time. We propose a solution that offers more options. Specifically, we do this by considering both pick-up time and price, so that travellers are able to choose the vehicle that matches their preferences best. In order to identify quickly vehicles that satisfy incoming ridesharing requests, we propose two efficient matching algorithms that follow the single-side and dual-side search paradigms, respectively. To further accelerate the matching, indexes on the road network and vehicles are developed, based on which several pruning heuristics are designed. Extensive experiments on a large Shanghai taxi dataset offer insights into the performance of our proposed techniques and compare with a baseline that extends the state-of-the art method.**

new scenario

I. Introduction

When travellers need to travel to some destination, they often have several options to choose among. In such a setting, the cost and travel time of an option are important decision criteria. Private cars and existing taxi services generally come at a higher cost, but may also be the fastest options. At the opposite side of the spectrum, fixed-line systems (e.g., buses), if available, may have lower cost, but may also be slower. Ridesharing holds the potential to achieve the convenience of private transportation and the low cost of collective transportation. Hence, ridesharing is attracting substantial attention, and deployments of ridesharing already exist (e.g., lyft [1], uberPOOL [2], etc.). Ridesharing refers to a transportation scenario where travellers with similar itineraries and time schedules share a vehicle for a trip and split the travel costs such as fuel, tolls, and parking fees. Advantages of ridesharing for the travellers, to society, and to the environment include reduced travel costs, travel time, congestion, fuel consumption, and air pollution.

Providing ridesharing at urban scale is non-trivial. A core issue is to develop a real-time matching algorithm that can quickly determine the best vehicle to satisfy incoming service requests. Existing real-time ridesharing systems [15], [17], [24] find only one option per request that tries to minimize system-wide vehicle travel time or distance. However, as illustrated by the following example, travellers may benefit from being offered several options. Assume that a couple has finished dinner at the seaside, which is far from the city center. They now want to travel home. Given that there are few nearby vehicles, getting a vehicle quickly is likely to cost extra because some vehicles must make a detour in order to pick them up. However, if they are willing to wait longer, they may pay less since some vehicles will be nearby later on. If we offer more options with different pick-up times and prices, travellers can choose the options they prefer. Hence, we develop a ridesharing solution that can offer several options per ridesharing request, where the options have different pick-up times and prices and where they do not dominate each other (option r_i dominates option r_j iff the pick-up time and price of r_i are earlier and lower than those of r_j [6]).

Although only one study [7] considers both pick-up time and price to return multiple options for each ridesharing request, it cannot be employed to solve our problem, due to the following two reasons. First, the problem definition is different. Cao et al. [7] assume that every vehicle has one pair of a start location and destination, and serves *only one* group of riders during a trip, which limits the usability and scalability of the ridesharing system. In our paper, we assume that the destination of a vehicle is not limited and that it can accommodate *any number* of rider groups during a trip as long as it satisfies a capacity constraint. Second, Cao et al. [7] use Euclidean distance for pruning, which is inefficient. In order to accelerate the matching between requests and vehicles, we present efficient index structures for both the road network and the trip schedules of vehicles, and develop corresponding pruning techniques to avoid unnecessary verifications. In brief, the key contributions of the paper are summarized as follows:

- We consider both the pick-up time and the price in dynamic ridesharing. In particular, our solution is able to return different options for each ridesharing request, where each option contains both the pick-up time and the price, and the options do not dominate each other.
- We propose efficient indexing for the road network and the trip schedules of vehicles. Based on the indexes, we also develop various pruning heuristics to improve the efficiency of the ridesharing.
- We present two efficient matching approaches, which follow the single-side search paradigm and the dual-side search paradigm, respectively. Specifically, the single-side search method proceeds from the start location of a ridesharing request, while the dual-side search method proceeds from the start location and the destination at the same time.

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00099

TABLE I
SYMBOLS AND DESCRIPTION

Notation	Description
$G\langle V, E, W\rangle$	A road network with a set V of vertices, a set E of edges, and an edge weight function W
u, v, or o	The vertex in a road network
g	The grid index of the road network
$dist(u, v)$	The shortest path distance between u and v
$ldist(u, v)$ or $udist(u, v)$	Lower or upper bound distance of $dist(u, v)$
$ldist(u, g)$	Lower bound distance of the minimal distance between the vertex u and the grid cell g
$tr = \langle o_1, o_2, \ldots, o_k\rangle$	A trip schedule contains a sequence of locations
$dist_{tr}$ or $dist_{tr}(u, v)$	Trip distance or trip distance between u and v
$ldist_{tr}(u, v)$ or $ldist_{tr'\text{-}tr}$	Lower bound distance of $dist_{tr}(u, v)$ and detour
$R = \langle s, d, n, w, \varepsilon\rangle$	A ridesharing request with a start location s, a destination location d, a number n of riders, a maximal waiting time w, and a service constraint ε
c	Empty or non-empty vehicle
$c.l$	Current location of c
$c.tr$ or $c.S_{tr}$	One or a set of valid trip schedule(s) of c
$\langle o_x, o_y\rangle$	Edge in the kinetic tree of $c.S_{tr}$
$o_x.capacity$, $o_x.detour$ or $o_x.dist_{tr}$	The capacity, the allowed detour distance, or the trip distance to $c.l$ stored in vertex o_x
$r = \langle c, time, price\rangle$ or $r = \langle c, dist_{pt}, price\rangle$	Result of a ridesharing request containing the vehicle c, the earliest pick-up time, and price
f_n	The price ratio

- We report extensive experiments on a large Shanghai taxi dataset, which offers insights into the performance of our proposed techniques and a baseline algorithm extended from the state-of-the-art method.

The rest of this paper is organized as follows. Section II reviews related work. Section III defines the ridesharing problem addressed. Sections IV and V detail index structures and corresponding pruning heuristics, respectively. Section VI presents a baseline algorithm and two matching algorithms. Empirical studies and our findings are covered in Section VII. Finally, Section VIII concludes and provides future directions.

II. Related Work

Ridesharing has recently attracted substantial attention, and existing ridesharing techniques are summarized well in surveys [3], [11]. The ridesharing mechanisms can be achieved by extending the dynamic pick-up-and-delivery [5], dial-a-ride [16], [26], and freight transportation methods [21]. In addition, multiple ridesharing systems [12], [14], [20] are developed. Recently, Huang et al. [17] propose kinetic tree algorithms that are better suited to efficient scheduling of dynamic requests and adjust routes on-the-fly. Ma et al. [15] develop a taxi-sharing system that accepts real-time ridesharing requests sent from smartphones and schedules taxis according to time, capacity, and monetary constraints. Thangaraj et al. [24] present a hierarchical discretization of geographical regions using grids, landmarks, and clusters with theoretical guarantees, along with efficient in-memory indexing of rides for maintaining spatio-temporal validity within a specified error tolerance. However, all the above proposals find for each rider just one choice that tries to minimize system-wide vehicle travel distance or time, which is different from our problem.

Cao et al. [7] consider both time cost and price. However, it is assumed that a vehicle has only one pair of a start location and a destination, and hence, it serves only one group of riders during its trip. These assumptions limit the usability and scalability of a ridesharing system. Duan et al. [10] explore personalized ridesharing, which considers user preferences with respect to time and price. Although these two studies consider time and price, their problems are different from ours.

In addition, orthogonal ridesharing studies exist that take into account the road network conditions [18], [27], privacy-preservation [22], [25], and pricing mechanisms [4], [28]. Motivated by ridesharing applications, Li et al. [19] propose a new type of query, namely, the optimal multi-meeting-point route query. Dai and Song et al. [9], [23] investigate the pick-up point recommendation. More recently, Cheng et al. [8] integrate rider satisfaction into a ridesharing system.

III. Problem Setting and Statement

In this section, we formalize the problem setting, and define the dynamic ridesharing problem. Table I summarizes the symbols used frequently throughout this paper.

A. Road Network

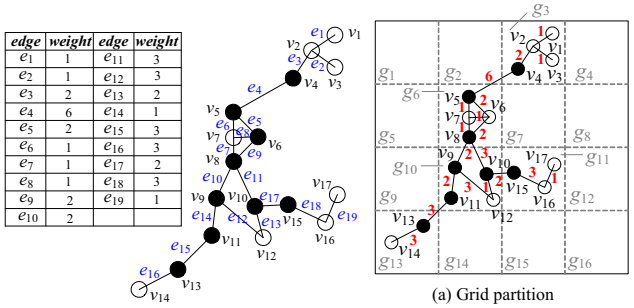
A road network $G = \langle V, E, W\rangle$ consists of a vertex set V and an edge set E. Each vertex $v \in V$ denotes the intersection of two roads. Each edge $e = (u, v) \in E$ that connects two vertices u and v is associated with a weight $W(e)$, where $W(e)$ denotes the travel cost between u and v. The travel cost could be either time or distance. When the speeds of vehicles are known, they can be converted from one to another (i.e., distance = time × speed). The road network is usually directed and $W(\ )$ is time dependent [13]. For simplicity, an undirected road network and a constant speed of vehicles are used in our experiments same as [17]. However, our methods can be easily extended to the directed road network and time dependent weights. Consider the road network in Fig. 1, where $V = \{v_1, \ldots, v_{17}\}$, $E = \{e_1, \ldots, e_{19}\}$, and the table lists the weight for each edge.

Given two vertices u and v in a road network, a path p between u and v is a vertex sequence $(o_1, o_2, \ldots, o_k)$, where $o_1 = u$, $o_k = v$, and $e_i = (o_i, o_{i+1})$. The path cost $W(p)$ is the sum of all the edge weights in the path. The shortest path distance between u and v (denoted as $dist(u, v)$) is defined as the minimal cost for paths linking between u and v. In Fig. 1, the shortest path between v_5 and v_8 is (v_5, v_7, v_8), and thus, $dist(v_5, v_8) = W(e_6) + W(e_7) = 2$. In this paper, we assume that each location (current, start, or destination location) is a vertex in the road network. However, our algorithms can be extended easily to locations that are not vertices, as a location l can be represented using the edge $e = (u, v)$ of the location with its distance $dist(l, u)$ or $dist(l, v)$ from one of the vertices.

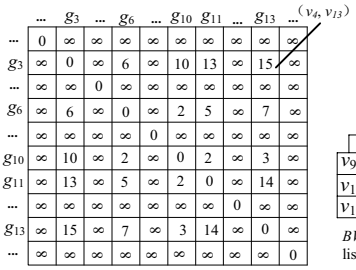
B. Ridesharing Request

Each ridesharing request can receive multiple options, where each option contains pick-up time and price. A rider can then choose the option that is preferred.

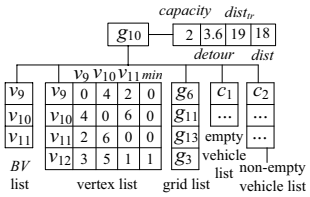
Definition 1 (*Ridesharing Request*). A ridesharing request $R = \langle s, d, n, w, \varepsilon\rangle$ on a road network G is defined by a start



(a) Grid partition



(b) Distance matrix



(c) Information of grid cell g_{10}

Fig. 1. Road network example

Fig. 2. Road network grid index example

location s, a destination location d, a number n of riders, a maximal waiting time w (the maximal time allowed between the planned pick-up time and the actual pick-up time), and a service constraint ε (the detour acceptable in a trip, i.e., the travel distance from s to d is bounded by $(1 + \varepsilon) \times dist(s, d)$).

The planned pick-up time is the pick-up time in a result returned when answering the ridesharing request R, while the actual pick-up time is the actual time when the riders in R get on the vehicle. The planned pick-up time is generally not equal to the actual one, because the vehicle may pick up other riders in a new ridesharing request R' before picking up riders in R. In Fig. 1, a ridesharing request R_1 is $\langle v_2, v_{16}, 2, 5, 0.2\rangle$, meaning that two riders want to travel from v_2 to v_{16} with a maximal waiting time of 5 and a service constraint of 0.2.

C. Vehicle Trip Schedule

Each vehicle can be assigned an empty or a non-empty set of unfinished ridesharing requests. A non-empty vehicle (i.e., with a non-empty set of unfinished ridesharing requests) has a set of valid vehicle trip schedules.

Definition 2 (*Vehicle Trip Schedule*). Each trip schedule $tr = \langle o_1, o_2, \ldots, o_k\rangle$ contains a sequence of locations (i.e., vertices in G), where o_1 is the current location $c.l$ of vehicle c, and o_i $(2 \le i \le k)$ denotes the start location or destination of an unfinished ridesharing request. The whole trip distance $dist_{tr}$ equals to $\sum_{i=1}^{k-1} dist(o_i, o_{i+1})$. A valid trip schedule needs to satisfy the four following conditions:

(1) **Capacity constraint:** At any time, the number of riders in a vehicle cannot exceed its capacity.

(2) **Point order:** For any unfinished ridesharing request $R = \langle s, d, n, w, \varepsilon\rangle$ in the vehicle trip schedule tr_i, the location of the vehicle that receives the request R must happen before the start location s, and the start location s must happen before its destination d.

(3) **Waiting time constraint:** For any unfinished request $R = \langle s, d, n, w, \varepsilon\rangle$ in the vehicle trip schedule tr_i, the waiting time between the planned and the actual pick-up time should not exceed the constraint w. Let tr_i be the actual trip schedule for vehicle c to pick up the rider, and let tr_j be the planned trip schedule when assigning R to c, $dist_{tr_i}(c.l, s) - dist_{tr_j}(c.l, s) \le w$.

(4) **Service constraint:** For any vehicle trip schedule tr, the actual travel distance from a start location s to a destination location d should not exceed the shortest distance between them multiplied by the service constraint, i.e., $dist_{tr}(s, d) \le (1 + \varepsilon) \times dist(s, d)$.

If we assign the request $R_1 = \langle v_2, v_{16}, 2, 5, 0.2\rangle$ to an empty vehicle c_1 (located at v_1), the resulting trip schedule $tr_1 = \langle v_1, v_2, v_{16}\rangle$ is valid, as it satisfies all the constraints. Then, we proceed to assign $R_2 = \langle v_{12}, v_{17}, 2, 5, 0.2\rangle$ to c_1, the resulting trip schedule $tr_2 = \langle v_1, v_2, v_{12}, v_{16}, v_{17}\rangle$ is valid, while the resulting trip schedule $tr_3 = \langle v_1, v_{12}, v_2, v_{16}, v_{17}\rangle$ is invalid, as it doesn't satisfy the waiting time constraint of R_1 (i.e., $dist_{tr_3}(v_1, v_2) - dist_{tr_1}(v_1, v_2) > 5$) and also doesn't satisfy the service constraint of R_2 (i.e., $dist_{tr_3}(v_2, v_{17}) > 1.2 \times dist(v_2, v_{17})$).

D. Price Model

As in an existing price model [7], the price of a request is the sum of prices for the detour and the original trip distances.

Definition 3 (*Price Model*). For a ridesharing request $R = \langle s, d, n, w, \varepsilon\rangle$, assume that tr_i is the current trip schedule of a vehicle c and tr_j is the new trip schedule after inserting R into tr_i, the price is computed as $f_n \times (dist_{tr_j} - dist_{tr_i} + dist(s, d))$, where f_n is price ratio that depends on the number n of riders.

We set $f_n = 0.3 + (n - 1) \times 0.1$. In Fig. 1, for a non-empty vehicle c_1 with the trip schedule $tr_1 = \langle v_1, v_2, v_{16}\rangle$, inserting the request R_2 into tr_1 results in a new $tr_2 = \langle v_1, v_2, v_{12}, v_{16}, v_{17}\rangle$, and hence, the corresponding price equals to $f_2 \times (dist_{tr_2} - dist_{tr_1} + dist(v_{12}, v_{17})) = 4$.

E. Dynamic Ridesharing

We proceed to give the definition of dynamic ridesharing.

Definition 4 (*Dynamic Ridesharing*). Given a set of vehicles C in the road network G at a specific time and a real-time trip request set S_R, dynamic ridesharing finds, for each trip request $R \in S_R$, all qualified and non-dominated results $\langle c, time, price\rangle$. Here, the result r_i dominates the result r_j iff $(r_i.time \le r_j.time \wedge r_i.price < r_j.price)$ or $(r_i.time < r_j.time \wedge r_i.price \le r_j.price)$.

Each vehicle c can offer one or more pairs of a pick-up time (i.e., *time*) and a corresponding *price* for R. Since the travel time can be transformed to the distance, we use the trip distance from c's current location l to R's start location s (termed $dist_{pt}$) to denote *time*. In the following, the result can also be denoted as $r_i = \langle c, dist_{pt}, price\rangle$.

In Fig. 1, assume that there exist two vehicles c_1 (with one trip schedule $tr_1 = \langle v_1, v_2, v_{16}\rangle$) and c_2 (with one trip schedule $tr_2 = \langle v_{13}\rangle$). Given the request $R_2 = \langle v_{12}, v_{17}, 2, 5, 0.2\rangle$, dynamic ridesharing returns the results $r_1 = \langle c_1, 14, 4\rangle$ and $r_2 = \langle c_2, 8, 8.8\rangle$, where r_1 has lower price, but r_2 has earlier pick-up time.

If multiple requests are issued at the same time, we take a greedy strategy, i.e., the ridesharing requests are processed one by one, due to two reasons. First, given a rigid real-time context, the system only has information on currently available ridesharing requests, which does not enable optimized schedules based on a global scope, i.e., over a long time span. Second, the problem of dynamic ridesharing for the complete ridesharing request stream is NP-complete [15].

IV. Index Structures

To efficiently answer dynamic ridesharing, we build the indexes on both the road network and each vehicle.

A. Road Network Indexing

We partition the road network using a grid, as illustrated in Fig. 2(a). We choose a grid index because it is simple yet efficient for dynamic ridesharing. Although other indexes (e.g., the Quadtree) that are adaptive to the network structure and density can improve the matching efficiency, their update costs are relatively higher for a dynamic road network. To this end, developing more efficient road network index with fast updates is left as a direction of our future work.

If an edge $e = (u, v)$ belongs to more than one grid cell, we call u and v **border vertices**. To estimate the shortest path distance precisely, we maintain a matrix M that stores the lower bound distance for every grid cell pair, as depicted in Fig. 2(b). Each entry D_{ij} in M is also associated with a vertex pair (x_{ij}, y_{ij}), where x_{ij} and y_{ij} are border vertices in the grid cells g_i and g_j, respectively, i.e., $(x_{ij}, v_1, v_2, \ldots, v_k, y_{ij})$ is the shortest path connecting g_i and g_j.

As depicted in Fig. 2(c), each grid cell maintains five lists: (i) a **border vertex list** of border vertices belonging to the grid cell; (ii) a **vertex list** of vertices belonging to the grid cell, as each vertex v is associated with its shortest path distances to the border vertices (BV) of the grid cell ($\{\langle u, dist(v, u)\rangle \mid u \in BV\}$) and is also associated with $min\{dist(v, u) \mid u \in BV\}$ (denoted as $v.min$); (iii) a **grid cell list** of other grid cells sorted in ascending order of the travel time from these grid cells to the grid cell (i.e., sorted in ascending order of the lower bound distances when the speed is constant); (iv) an **empty vehicle list** of the vehicles with empty ridesharing requests in g_i; and (v) a **non-empty vehicle list** of vehicles c_j with a non-empty set of ridesharing requests, i.e., vehicles c_j are currently located in g_j or are scheduled to enter g_i.

Based on the road network indexing, we can derive the lower and upper bounds on shortest path distances quickly.

Definition 5 (*Lower Bound Distance*). Given a vertex u in a grid cell g_i and a vertex v in a grid cell g_j, if $g_i \neq g_j$, then the lower bound distance $ldist(u, v) = D_{ij} + u.min + v.min$; if $g_i = g_j$, then $ldist(u, v) = 0$.

Definition 6 (*Upper Bound Distance*). Given a vertex u in a grid cell g_i and a vertex v in g_j, if $g_i \neq g_j$, then the upper bound distance $udist(u, v) = D_{ij} + dist(u, x_{ij}) + dist(v, y_{ij})$; if $g_i = g_j$, then $udist(u, v) = min\{dist(u, b) + dist(v, b) \mid b \in BV\}$.

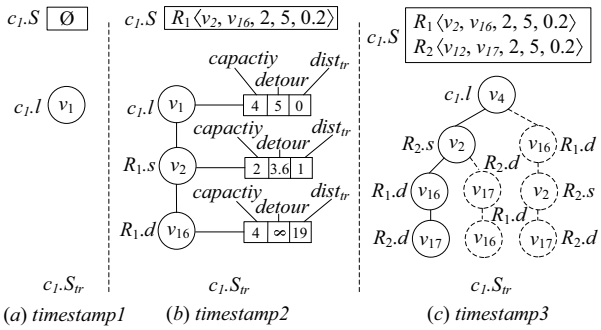


Fig. 3. Kinetic tree examples on vehicle trip schedules at three timestamps

Consider the example shown in Fig. 2(a), $ldist(v_2, v_{12}) = 13$ and $udist(v_2, v_{12}) = 15$, which are tight bounds of $dist(v_2, v_{12}) = 14$. As values used to compute *ldist* and *udist* are stored in the grid index, we can estimate the shortest path distance within O(1) or O($|BV|$) to avoid traversing the road network.

B. Index of Vehicle Trip Schedules

Each vehicle c is represented using (i) the unique identifier $c.ID$ of the vehicle; (ii) the current location $c.l$ of the vehicle; (iii) the set $c.S$ of unfinished ridesharing requests assigned to it, sorted in ascending order of their timestamps; and (iv) the set $c.S_{tr}$ of all valid vehicle trip schedules.

The set $c.S_{tr}$ of all valid vehicle trip schedules can be managed by a kinetic tree [17], where every branch in a kinetic tree denotes a valid trip schedule. For example, in Fig. 3(c), given two unfinished requests R_1 and R_2 assigned to vehicle c_1, the first branch $\langle v_4, v_{12}, v_{16}, v_{17}\rangle$ of the kinetic tree denotes a possible trip schedule, where c_1 first picks up the riders of R_2, then drops off the riders of R_1, and finally drops off the riders of R_2. In addition, we add three variables to each node o_x of the kinetic tree: (i) a current vehicle's *capacity*, (ii) the minimal *detour* distance allowed, and (iii) the trip distance $dist_{tr}$ from c's current location to o_x. The *detour* can be computed easily from the waiting time and service constraints. For instance, in Fig. 3(b), (i) the waiting time constraint of R_1 is 5, and thus, the *detour* allowed in $\langle v_1, v_2\rangle$ is 5, i.e., $v_1.detour = 5$; and (ii) the service constraint of R_1 is 0.2, and hence, the *detour* allowed in $\langle v_2, v_{16}\rangle$ is $0.2 \times dist(v_2, v_{16}) = 3.6$, i.e., $v_2.detour = 3.6$. If an edge $\langle o_x, o_y\rangle$ of vehicle c's kinetic tree intersects with, or is contained in a grid cell g_i (i.e., the shortest path between o_x and o_y intersects with or is contained in g_i), then $\langle o_x, o_y\rangle$ belongs to the non-empty vehicle list of g_i.

Consider the index example of vehicle c_1 depicted in Fig. 3, where ridesharing requests $R_1 = \langle v_2, v_{16}, 2, 5, 0.2\rangle$ and $R_2 = \langle v_{12}, v_{17}, 2, 5, 0.2\rangle$ are issued in turn. At *timestamp2*, the node v_1 is associated with $v_1.capacity = 4$, the minimum $v_1.detour$ distance is 5 (derived according to the waiting time constraint of R_1), and the trip distance $v_1.dist_{tr}$ is $dist_{tr}(v_1, v_1) = 0$; and the node $R_1.s$ (= v_2) is associated with $v_2.capacity = 2$, $v_2.detour = 3.6$ (derived according to the service constraint of R_1), and the trip distance $v_2.dist_{tr}$ is $dist_{tr}(v_1, v_2) = 1$.

To facilitate the pruning of a whole grid cell (to be discussed in Section V), we also maintain the following global information of each grid cell g_j: (i) the maximum ***capacity*** = $max\{o_x.capacity \mid \langle o_x, o_y\rangle \in g_j\}$; (ii) the maximal ***detour*** = $max\{o_x.detour \mid \langle o_x, o_y\rangle \in g_j\}$ allowed, (iii) the minimal trip distance ***dist$_{tr}$*** = $min\{o_x.dist_{tr} \mid \langle o_x, o_y\rangle \in g_j\}$, and (iv) the maximal shortest path distance ***dist*** = $max\{dist(o_x, o_y) \mid \langle o_x, o_y\rangle \in g_j\}$. Note that, $dist(o_x, o_y)$ is computed as $dist_{tr}(c.l, o_y) - dist_{tr}(c.l, o_x)$, while $dist_{tr}(c.l, o_x)$ and $dist_{tr}(c.l, o_y)$ are stored as $o_x.dist_{pt}$ and $o_y.dist_{pt}$ in the tree. Fig. 2(c) exemplifies the global information of grid cell g_{10} at *timestamp2* in Fig. 3(b).

C. Discussion

We first analyze the space complexity of indexes, and then, discuss how to update the indexes at every time unit.

Let n_g be the number of grid cells, n_g' be the number of non-empty grid cells, n_b be the average number of the border vertices in each grid cell, n_v be the number of vehicles, n_r be the average number of the unfinished requests assigned to each vehicle, and n_e be the average number of the grid cells that intersect with each kinetic tree edge $\langle o_x, o_y\rangle$.

By definition, each branch of a kinetic tree is a possible trip schedule of all the unfinished requests. In the worst case, $(2n_r)!$ trip schedules exist for every vehicle. However, the actual number of branches is much lower than $(2n_r)!$ due to the constraints in Definition 2. Hence, the space complexity for all the kinetic trees is $O(n_v \times n_r \times (2n_r)!)$, and the space complexity for the grid index is $O(n_g^2 + n_g'^2 + n_b \times n_g' + |V| \times (n_b+1) + n_e \times n_v \times n_r \times (2n_r)!)$. As observed, the size of the kinetic trees is related to the request distribution and the number of vehicles, while the size of the grid index is related to the number of grid cells, the distribution of the road network, and the kinetic tree size.

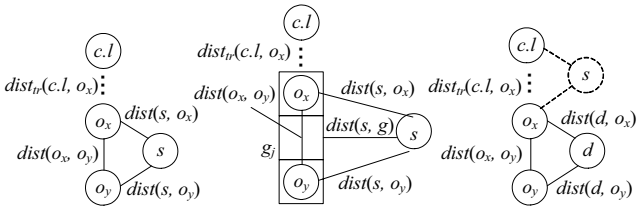
The current location of each vehicle is updated at every time unit, resulting in an index update. (i) The nodes connected to the root node in each kinetic tree are updated. The time complexity for updating kinetic trees is $O(n_v \times (2n_r)!)$. (ii) The empty and non-empty vehicle lists related to each kinetic tree are also updated. The time complexity for updating the grid index is $O(n_e \times n_v \times (2n_r)!)$.

V. Pruning Heuristics

Existing work utilizes the waiting time constraint to filter unqualified vehicles. Our dynamic ridesharing problem trades time for price. Hence, the waiting time constraint cannot be used directly for filtering vehicles. To enable filtering and thus reduce the computational costs, based on the designed road network and vehicle indexes, we use the lower and upper bound distances instead of the actual shortest path distances to estimate pick-up time and price for filtering

A. Pruning Empty Vehicles

Using a grid index, we can estimate the lower and upper bounds of the pick-up time and price returned by an empty vehicle c_j. Hence, given a current valid result r_i, we try to derive (i) pruning conditions implying that the estimated result r_j (i.e., the lower bound values) returned by c_j is dominated by r_i; and (ii) the pruning conditions implying that r_j (i.e., the upper bound values) returned by c_j dominates r_i.



(a) Lemma 3 (b) Lemma 4 (c) Lemma 9

Fig. 4. Illustration of Lemmas 3, 4, and 9

Lemma 1. Given a ridesharing request $R = \langle s, d, n, w, \varepsilon\rangle$, a result $r_i = \langle c, dist_{pt}, price\rangle$ for R, and an empty vehicle c_j, if $ldist(c_j.l, s) > max\{r_i.dist_{pt}, r_i.price/f_n - 2 \times dist(s, d)\}$, c_j can be pruned safely; and if $udist(c_j.l, s) < min\{r_i.dist_{pt}, r_i.price/f_n - 2 \times dist(s, d)\}$, r_i can be removed from the result set.

Proof. Assume that the result $r_j = \langle c, dist_{pt}, price\rangle$ returns after inserting R into the trip schedule of an empty vehicle c_j. If $ldist(c_j.l, s) > max\{r_i.dist_{pt}, r_i.price/f_n - 2 \times dist(s, d)\}$, then $r_j.dist_{pt} = dist(c_j.l, s) \geq ldist(c_j.l, s) > r_i.dist_{pt}$ and $r_j.price = f_n \times (dist(c_j.l, s) + 2 \times dist(s, d)) \geq f_n \times (ldist(c_j.l, s) + 2 \times dist(s, d)) > r_i.price$. Hence, r_i dominates r_j, and thus c_j can be pruned safely. Similarly, if $udist(c_j.l, s) < min\{r_i.dist_{pt}, r_i.price/f_n - 2 \times dist(s, d)\}$, then $r_j.dist_{pt} \leq udist(c_j.l, s) < r_i.dist_{pt}$, and $r_j.price \leq f_n \times (udist(c_j.l, s) + 2 \times dist(s, d)) < r_i.price$. Hence, r_j dominates r_i, and thus r_i can be removed from the result set. □

Lemma 1 is applied for a single vehicle. Next, we develop a lemma to prune the whole empty vehicle list of a grid cell.

Lemma 2. Given a ridesharing request $R = \langle s, d, n, w, \varepsilon\rangle$, a result $r_i = \langle c, dist_{pt}, price\rangle$ for R, and a grid cell g_j, if $ldist(g_j, s) > max\{r_i.dist_{pt}, r_i.price/f_n - 2 \times dist(s, d)\}$, the empty-vehicle list of g_j can be pruned safely.

Proof. The proof is omitted due to similar as Lemma 1. □

Consider the road network shown in Fig. 2(a). Given a ridesharing request $R_1 = \langle v_2, v_{16}, 2, 5, 0.2\rangle$ and a result $r_i = \langle c_1, 1, 14.8\rangle$ for R_1, all the empty vehicles located in grid cells g_6, g_{10}, g_{11}, and g_{13} can be pruned by r_i due to Lemmas 1 and 2.

B. Pruning Non-Empty Vehicles

For non-empty vehicles, we insert the start location and destination into the original trip schedules, respectively. In the sequel, we develop pruning lemmas for these two phases.

1) Pruning for inserting start location *s*

When inserting s into $\langle o_x, o_y\rangle$ of a valid trip schedule $c_j.tr$, we can quickly compute the lower bounds in the pick-up time and price. If the estimated result (i.e., the lower bounds) returned by c_j is dominated by a current valid result, we cannot insert s into $\langle o_x, o_y\rangle$. Hence, given a current result r_i, we try to derive the pruning conditions implying that the new result r_j returned by c_j is dominated by r_i.

Lemma 3. Given a request $R = \langle s, d, n, w, \varepsilon\rangle$, a result $r_i = \langle c_i, dist_{pt}, price\rangle$ for R, and a non-empty vehicle c_j with $\langle o_x, o_y\rangle$ in its valid trip schedule $c_j.tr$, if $ldist(s, o_x) + dist_{tr}(c_j.l, o_x) > r_i.dist_{pt}$ and $ldist(s, o_x) + ldist(s, o_y) - dist(o_x, o_y) > r_i.price/f_n -$

$dist(s, d)$, $\langle o_x, o_y \rangle$ is pruned for the non-empty vehicle c_j (i.e., the start location s cannot be added to $\langle o_x, o_y \rangle$ of $c_j.tr$).

Proof. As shown in Fig. 4(a), assume that we insert s into $\langle o_x, o_y \rangle$ of a non-empty vehicle's valid trip schedule $c_j.tr$, and that returns a result $r_j = \langle c_j, dist_{tr}, price \rangle$. If $ldist(s, o_x) + dist_{tr}(c_j.l, o_x) > r_i.dist_{pt}$, then $r_j.dist_{pt} = dist_{tr}(c_j.l, o_x) + dist(s, o_x) \geq dist_{tr}(c_i.l, o_x) + ldist(s, o_x) > r_i.dist_{pt}$. If $ldist(s, o_x) + ldist(s, o_y) - dist(o_x, o_y) \geq r_i.price/f_n - dist(s, d)$, then $r_j.price \geq f_n \times (ldist(s, o_x) + ldist(s, o_y) - dist(o_x, o_y) + dist(s, d)) > r_i.price$. Therefore, r_i dominates r_j, and, s cannot be inserted into $\langle o_x, o_y \rangle$ of $c_j.tr$. □

Back to the example in Fig. 3(b), given a non-empty vehicle c_1 with a trip schedule $tr_i = \langle v_1, v_2, v_{16} \rangle$, a ridesharing request $R_2 = \langle v_{12}, v_{17}, 2, 5, 0.2 \rangle$, and a result $\langle c_2, 8, 8.8 \rangle$, then $\langle v_1, v_2 \rangle$ can be discarded, while $\langle v_2, v_{16} \rangle$ cannot be pruned according to Lemma 3. Note that, for $\langle o_x, o_y \rangle$, o_x can be the last node in the trip schedule, i.e., $\langle v_{16}, \varnothing \rangle$ in the example. Hence, if o_y is empty, then $ldist(s, o_x) + ldist(s, o_y) - dist(o_x, o_y)$ is modified to $ldist(s, o_x) + 2 \times dist(s, d)$. Next, similar as Lemma 3, we try to prune the whole grid cell when inserting s.

Lemma 4. Given a ridesharing request $R = \langle s, d, n, w, \varepsilon \rangle$, a result $r_i = \langle c_i, dist_{pt}, price \rangle$ for R, and a grid cell g_j, if $ldist(s, g_j) + min\{o_x.dist_{tr} \mid \langle o_x, o_y \rangle \in g_j\} > r_i.dist_{pt}$ and $2 \times ldist(s, g_j) - max\{dist(o_x, o_y) \mid \langle o_x, o_y \rangle \in g_j\} > r_i.price/f_n - dist(s, d)$, we can prune all kinetic tree edges $\langle o_x, o_y \rangle$ contained in g_j.

Proof. As depicted in Fig. 4(b), for each $\langle o_x, o_y \rangle \in g_j$, we can get that $dist(s, o_x) \geq dist(s, g_j) \geq ldist(s, g_j)$ and $dist(s, o_y) \geq dist(s, g_j) \geq ldist(s, g_j)$, because the grid cells are processed in ascending order of their minimal distances to the start location s, and each $\langle o_x, o_y \rangle$ is processed only once. Assume that for any $\langle o_x, o_y \rangle \in g_j$, a result $r_j = \langle c_j, dist_{tr}, price \rangle$ is returned by inserting s in $\langle o_x, o_y \rangle$. For each returned r_j, if $ldist(s, g_j) + min\{o_x.dist_{tr} \mid \langle o_x, o_y \rangle \in g_j\} > r_i.dist_{pt}$, then $r_j.dist_{pt} = dist_{tr}(c_j.l, o_x) + dist(s, o_x) \geq min\{o_x.dist_{tr} \mid \langle o_x, o_y \rangle \in g_j\} + ldist(s, g_j) > r_i.dist_{pt}$; and if $2 \times ldist(s, g_j) - max\{dist(o_x, o_y) \mid \langle o_x, o_y \rangle \in g_j\} > r_i.price/f_n - dist(s, d)$, then $r_j.price \geq f_n \times (ldist(s, g_j) + ldist(s, g_j) - max\{dist(o_x, o_y) \mid \langle o_x, o_y \rangle \in g_j\} + dist(s, d)) > r_i.price$. Hence, r_i dominates every returned r_j, and thus, we can prune all kinetic tree edges $\langle o_x, o_y \rangle$ contained in g_j. □

Note that, $min\{o_x.dist_{tr} \mid \langle o_x, o_y \rangle \in g_j\}$ and $max\{dist(o_x, o_y) \mid \langle o_x, o_y \rangle \in g_j\}$ used in Lemma 4 are stored in the grid index of the road network. In the following, we utilize the capacity constraint and *detour* to prune unqualified vehicles.

Lemma 5. Given a request $R = \langle s, d, n, w, \varepsilon \rangle$ and a non-empty vehicle c_j with $\langle o_x, o_y \rangle$ in its valid trip schedule $c_j.tr$, if $o_x.capacity < n$ or $o_x.detour < ldist(s, o_x) + ldist(s, o_y) - dist(o_x, o_y)$, we can prune $\langle o_x, o_y \rangle$ for the non-empty vehicle c_j.

Proof. It can be easily proved by the capacity constraint and the definition of *detour* stored in the kinetic tree. □

In Fig. 3(b), the detour distance allowed in node v_1 is 5, then $\langle v_1, v_2 \rangle$ can be pruned due to $v_1.detour < ldist(v_{12}, v_1) + ldist(v_{12}, v_2) - dist(v_1, v_2)$. Next, we also try to discard the whole grid cell using the capacity and service constraints.

Lemma 6. Given a request $R = \langle s, d, n, w, \varepsilon \rangle$ and a grid cell g_j, if $max\{o_x.capacity \mid \langle o_x, o_y \rangle \in g_j\} < n$ or $max\{o_x.detour \mid \langle o_x, o_y \rangle \in g_j\} < 2 \times ldist(s, g_j) - max\{dist(o_x, o_y) \mid \langle o_x, o_y \rangle \in g_j\}$, we can prune all kinetic tree edges $\langle o_x, o_y \rangle$ in g_j.

Proof. The proof is omitted due to similar as Lemma 5. □

Note that, for Lemmas 5 and 6, if we insert s after the last node of a trip schedule, i.e., o_y is empty, the detour constraint cannot be used for pruning.

2) Pruning for inserting destination location *d*

As in Lemma 5, the detour distance can also be used for pruning when inserting destination d.

Lemma 7. Given a ridesharing request $R = \langle s, d, n, w, \varepsilon \rangle$, a non-empty vehicle c_j with $\langle o_x, o_y \rangle$ in its valid trip schedule $c_j.tr$, if $o_x.capacity < n$ or $o_x.detour < ldist(d, o_x) + ldist(d, o_y) - dist(o_x, o_y)$, we can prune $\langle o_x, o_y \rangle$ of $c_j.tr$, i.e., we cannot insert d in $\langle o_x, o_y \rangle$ of $c_j.tr$.

Proof. It can be easily proved by the capacity constraint and the definition of *detour* stored in the kinetic tree. □

Back to the example in Fig. 3(b), given a new ridesharing request $R_2 = \langle v_{12}, v_{17}, 2, 5, 0.2 \rangle$, then $\langle v_1, v_2 \rangle$ can be pruned while $\langle v_2, v_{16} \rangle$ cannot be pruned by Lemma 7. Similar as inserting s, if we insert d after the last node of a trip schedule, i.e., o_y is empty, Lemma 7 cannot be used for pruning. Next, we try to prune all the edges $\langle o_x, o_y \rangle$ in the grid cell.

Lemma 8. Given a request $R = \langle s, d, n, w, \varepsilon \rangle$ and a grid cell g_j, if $max\{o_x.capacity \mid \langle o_x, o_y \rangle \in g_j\} < n$ or $max\{o_x.detour \mid \langle o_x, o_y \rangle \in g_j\} < 2 \times ldist(d, g_j) - max\{dist(o_x, o_y) \mid \langle o_x, o_y \rangle \in g_j\}$, we can prune all kinetic tree edges $\langle o_x, o_y \rangle$ contained in g_j.

Proof. The proof is omitted due to similar as Lemma 7. □

In addition, using the service constraint, we can derive the lower bounds of the pick-up time and the price when inserting the destination into $c_j.tr$. Thus, given a current result r_i, we strive to derive the pruning conditions implying that the new result r_j (lower bound values) returned by c_j is dominated by r_i.

Lemma 9. Given a request $R = \langle s, d, n, w, \varepsilon \rangle$, a result $r_i = \langle c_i, dist_{pt}, price \rangle$ for R, and a non-empty vehicle c_j with $\langle o_x, o_y \rangle$ in its valid trip schedule tr, if $dist_{tr}(c_j.l, o_x) + ldist(o_x, d) - (1 + \varepsilon) \times dist(s, d) > r_i.dist_{pt}$ and $ldist(o_x, d) + ldist(o_y, d) - dist(o_x, o_y) > r_i.price/f_n - dist(s, d)$, we can prune $\langle o_x, o_y \rangle$ in $c_j.tr$ for the non-empty vehicle c_j, i.e., the destination location d cannot be inserted in $\langle o_x, o_y \rangle$ belonged to $c_j.tr$.

Proof. Assume that we insert d in $\langle o_x, o_y \rangle$ of a non-empty vehicle's valid trip schedule $c_j.tr$, and that returns a result $r_j = \langle c_j, dist_{pt}, price \rangle$, as illustrated in Fig. 4(c). If $dist_{tr}(c_j.l, o_x) + ldist(o_x, d) - (1 + \varepsilon) \times dist(s, d) > r_i.dist_{pt}$ and $ldist(o_x, d) + ldist(o_y, d) - dist(o_x, o_y) > r_i.price/f_n - dist(s, d)$, we can get that $r_j.dist_{tr} \geq dist_{tr}(c_j.l, o_x) + ldist(o_x, d) - (1 + \varepsilon) \times dist(s, d) > r_i.dist_{pt}$ due to the service constraint, and $r_j.price \geq f_n \times (ldist(d, o_x) + ldist(d, o_y) - dist(o_x, o_y) + dist(s, d)) > r_i.price$. Therefore, r_i dominates r_j, and d cannot be added to $\langle o_x, o_y \rangle$ of $c_j.tr$. □

Back to the example depicted in Fig. 3(b), given a result $r_i = \langle c_2, 4, 3 \rangle$ for R_2, if we insert the destination v_{17} of R_2 into $\langle v_2,$

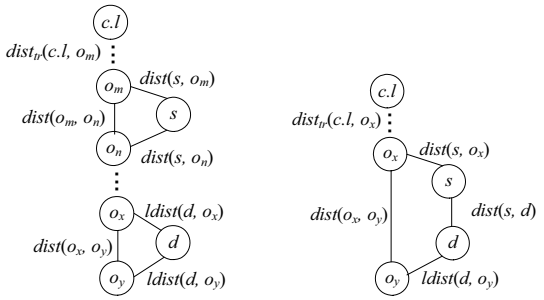


Fig. 5. Illustration of Definition 7

$v_{16}\rangle$, then $\langle v_2, v_{16}\rangle$ can be pruned using Lemma 9 since $v_2.dist_{tr} + ldist(v_2, v_{12}) - 1.2 \times dist(v_{12}, v_{17}) > 4$ and $ldist(v_2, v_{12}) + ldist(v_{16}, v_{12}) - dist(v_2, v_{16}) > 3/f_2 - dist(v_{12}, v_{17})$.

Lemma 10. Given a request $R = \langle s, d, n, w, \varepsilon\rangle$, a result $r_i = \langle c_i, dist_{pt}, price\rangle$ for R, and a grid cell g_j, if $min\{o_x.dist_{tr} \mid \langle o_x, o_y\rangle \in g_j\} + ldist(g_j, d) - (1 + \varepsilon) \times dist(s, d) > r_i.dist_{pt}$ and $2 \times ldist(d, g_j) - max\{dist(o_x, o_y) \mid \langle o_x, o_y\rangle \in g_j\} > r_i.price/f_n - dist(s, d)$, we can prune all kinetic tree edges $\langle o_x, o_y\rangle$ in g_j.

Proof. The proof is omitted due to similar as Lemma 9. □

Since we insert start location s into the trip schedule (i.e., the actual distances related to s are already computed after inserting s) before inserting destination d, we can derive a lower bound of the detour for the new result when inserting d.

Definition 7 (*Lower Bound of Detour*). Given a ridesharing request $R = \langle s, d, n, w, \varepsilon\rangle$, a non-empty vehicle c, and a valid trip schedule $c.tr = \langle o_1, o_2, \ldots, o_k\rangle$, assume that s is inserted into $\langle o_m, o_n\rangle \in c.tr$ and d is to be inserted into $\langle o_x, o_y\rangle \in c.tr$. This leads to a new trip schedule $c.tr'$, as shown in Fig. 5. A lower bound on detour between $c.tr'$ and $c.tr$ is computed as:

(1) If $\langle o_m, o_n\rangle \neq \langle o_x, o_y\rangle$, then $ldist_{tr'\text{-}tr} = dist(o_m, s) + dist(o_n, s) - dist(o_m, o_n) + ldist(o_x, d) + ldist(o_y, d) - dist(o_x, o_y)$.

(2) If $\langle o_m, o_n\rangle = \langle o_x, o_y\rangle$, then $ldist_{tr'\text{-}tr} = dist(o_x, s) + ldist(o_y, d) + dist(s, d) - dist(o_x, o_y)$.

If o_y is an empty node, i.e., d is inserted into the end of $c.tr$, then $dist(o_x, o_y) = ldist(o_y, d) = 0$. In the example depicted in Fig. 3(b), if we insert the start location v_{12} and the destination v_{17} of R_2 into $\langle v_2, v_{16}\rangle$, then we can get that $ldist_{tr'\text{-}tr} = dist(v_{12}, v_2) + ldist(v_{17}, v_{16}) + dist(v_{12}, v_{17}) - dist(v_2, v_{16}) = 2$.

As s is inserted, the pick-up time is already known. We then derive conditions for the lower bound of the detour so that the new result returned is dominated by a current result.

Lemma 11. Given a request $R = \langle s, d, n, w, \varepsilon\rangle$, a result $r_i = \langle c_i, dist_{pt}, price\rangle$ for R, a non-empty vehicle c_j, and a valid trip schedule $c_j.tr$ in $c_j.S_{tr}$, assume that $c_j.tr'$ is a new trip schedule after inserting R into $c_j.tr$. If $dist_{tr'}(c_j.l, s) > r_i.dist_{pt}$ and $ldist_{tr'\text{-}tr} > r_i.price/f_n - dist(s, d)$, then $c_j.tr$ can be pruned.

Proof. Assume that $c_j.tr'$ is a valid trip schedule obtained by inserting R into $c_j.tr$, and that returns the result $r_j = \langle c_j, dist_{pt}, price\rangle$. If $dist_{tr'}(c_j.l, s) > r_i.dist_{pt}$ and $ldist_{tr'\text{-}tr} > r_i.price/f_n - dist(s, d)$, then $r_j.dist_{pt} = dist_{tr'}(c_j.l, s) > r_i.dist_{pt}$ and $r_j.dist_{pt} \geq f_n (ldist_{tr'\text{-}tr} + dist(s, d)) > r_i.price$. Consequently, r_j is dominated by r_i, and $c_j.tr$ can be pruned. □

Algorithm 1 Single-Side Search Algorithm (SSA)

Input: a ride request $R = \langle s, d, n, w, \varepsilon\rangle$, a grid index on a road network G
Output: the result set S_r for R
/* S_c: a set of candidate vehicles */
1: $S_r = \emptyset$, $S_c = \emptyset$, and $S_g = \{g_i\} \cup g_i.L_g$ // s locates in a grid cell g_i
2: **for** each grid cell g_j in S_g **do**
3: $S_c = find\_empty\_vehicle\ (g_j, S_r)$ // Algorithm 2
4: $S_c = S_c \cup find\_nonempty\_vehicle\ (g_j, S_r)$ // Algorithm 3
5: **for** each vehicle c *in* S_c **do**
6: $find\_result(R, c, S_r)$ // Algorithm 4
7: **return** S_r

In the example depicted in Fig. 3(b), given a result $r_i = \langle c_2, 8, 8.8\rangle$ for R_2, if we insert the start location v_{12} and the destination location v_{17} of R_2 into $\langle v_2, v_{16}\rangle$ of $c_1.tr$, then $c_1.tr$ cannot be discarded by Lemma 11 because $dist_{tr'}(c_1.l, v_{12}) > r_i.dist$ and $ldist_{tr'\text{-}tr} < r_i.price/f_n - dist(s, d)$.

VI. Ridesharing Matching Algorithms

In this section, we present matching algorithms to solve our dynamic ridesharing problem. The matching algorithms try to find all qualified answers for every request.

A. Baseline Algorithm

A Baseline Algorithm can be obtained from the kinetic tree algorithm [17]. Specifically, for every vehicle, we insert each request $R = \langle s, d, n, w, \varepsilon\rangle$ into its kinetic tree, and return all possible pairs of pick-up time and price that cannot dominate each other. This procedure can be improved in two ways: (i) we do not need to verify all vehicles, i.e., we can filter out unqualified vehicles in advance; and (ii) the number of the shortest path distance computations ([17] computes all the distances before verification) can be reduced when inserting R into the kinetic tree, resulting in two algorithms presented next.

B. Single-Side Search Algorithm

For each ridesharing request $R = \langle s, d, n, w, \varepsilon\rangle$, we start searching for vehicles in the grid cell g_i where s is located, and then we search in the grid cell list of g_i (i.e., other grid cells sorted in ascending order of their minimal distances to g_i). This is because, the closer a vehicle is to s, the earlier it will pick up the riders of R. During the search, Lemmas 1 and 2 are used to avoid unnecessary verifications of empty vehicles, while Lemmas 3 to 6 are utilized to filter unqualified non-empty vehicles. For every empty vehicle not pruned, R can be inserted easily into its trip schedule set. For every non-empty vehicle not pruned, the inserting proceeds as described in the literature [17], by integrating Lemmas 3, 5, 7, and 11 to avoid unnecessary shortest path distance computations. Specifically, we first utilize Lemmas 3 and 5 to prune unqualified positions to insert the start location s into the kinetic tree. Note that, Lemmas 3 and 5 only employ the lower bound distances for pruning. If the positions cannot be pruned, we insert s by computing the actual shortest path distances. Then, we use Lemmas 7 and 11 to prune unqualified positions, to insert the destination d into the kinetic tree. We insert d into qualified

Algorithm 2 Find_Empty_Vehicle Algorithm

Input: a grid cell g_j, a result set S_r
Output: a set S_c of empty candidate vehicles
1: $S_c = \emptyset$, *flag* = true, *dominate* = true
2: **if** $S_r \neq \emptyset$ **then**
3: **for** each $r_i = \langle c, dist_{pt}, price \rangle$ in S_r **do**
4: **if** $ldist(g_j, s) > max\{r_i.dist_{pt}, r_i.price/f_n - 2 \times dist(s, d)\}$ **then**
5: *flag* = false and **break** // Lemma 2
6: **if** *flag* **then** // verify empty vehicles
7: **for** each vehicle $c_k \in g_j.L_e \wedge c_k$ is not verified **do**
8: *dominate* = true
9: **for** each $r_i = \langle c, dist_{pt}, price \rangle$ in S_r **do**
10: **if** $ldist(c_k.l, s) > max\{r_i.dist_{pt}, r_i.price/f_n - 2 \times dist(s, d)\}$ **then**
11: *dominate* = false and **break** // Lemma 1
12: **if** *dominate* **then** *insert* c_k *into* S_c
13: **return** S_c

Algorithm 3 Find_Nonempty_Vehicle Algorithm

Input: a grid cell g_j, a result set S_r
Output: a set S_c of empty candidate vehicles
1: $S_c = \emptyset$, *flag* = true, *dominate* = true
2: **if** $max\{o_x.capacity \mid \langle o_x, o_y \rangle \in g_j\} < n$ or $max\{o_x.detour \mid \langle o_x, o_y \rangle \in g_j\} < 2 \times ldist(s, g_j) - max\{dist(o_x, o_y) \mid \langle o_x, o_y \rangle \in g_j\}$ **then**
3: **return** S_c // Lemma 6
4: **if** $S_r \neq \emptyset$ **then**
5: **for** each $r_i = \langle c, dist_{pt}, price \rangle$ in S_r **do**
6: **if** $ldist(s, g_j) + min\{o_x.dist_{tr} \mid \langle o_x, o_y \rangle \in g_j\} > r_i.dist_{pt} \wedge 2 \times ldist(s, g_j) - max\{dist(o_x, o_y) \mid \langle o_x, o_y \rangle \in g_j\} > r_i.price/f_n - dist(s, d)$ **then**
7: *flag* = false and **break** // Lemma 4
8: **if** *flag* **then** // verify non-empty vehicles
9: **for** each $\langle o_x, o_y \rangle$ of $c_k.tr \in g_i.L_n$ and c_k is not verified **do**
10: *dominate* = true
11: **if** $o_x.capacity < n$ or $o_x.detour < ldist(s, o_x) + ldist(s, o_y) - dist(o_x, o_y)$ **then**
12: *dominate* = false and **break** // Lemma 5
13: **for** each $r_i = \langle c_i, dist_{pt}, price \rangle$ in S_r **do**
14: **if** $ldist(s, o_x) + dist_{tr}(c_k.l, o_x) > r_i.dist_{pt} \wedge ldist(s, o_x) + ldist(s, o_y) - dist(o_x, o_y) > r_i.price/f_n - dist(s, d)$ **then**
15: *dominate* = false and **break** // Lemma 3
16: **if** *dominate* **then** insert c_k into S_c
17: **return** S_c

positions not pruned, and calculate the real distances related to d. Since s is already inserted into the tree before inserting d, Lemma 11 uses the actual distances from s to other locations.

Algorithm 1 depicts the pseudo-code of Single-Side Search Algorithm (SSA), which takes as inputs a request $R = \langle s, d, n, w, \varepsilon \rangle$ and a grid index on a road network, and outputs the result set S_r for R. First, SSA initializes the result set S_r and the vehicle candidate set S_c as empty, and it sets the grid cell set S_g to $\{g_i\} \cup g_i.L_g$ (line 1). Here, g_i is the current grid cell that contains the start location s of R, and $g_i.L_g$ contains the other grid cells sorted in ascending order of their minimal distances to g_i. Then, the algorithm verifies the grid cells in S_g. For each grid cell g_j in S_g, it calls *find_empty_vehicle* algorithm and *find_nonempty_vehicle* algorithm to find the candidate vehicle set S_c in this grid cell (lines 3-4), and then, for each candidate vehicle c in S_c, it invokes *find_result* to insert the ridesharing request R into c's trip schedules (lines 5-6). Finally, SSA returns the result set S_r (line 7).

Algorithm 2 shows the pseudo-code of *find_empty_vehicle*. It first initializes the vehicle candidate set S_c as empty, and sets *flag* and *dominate* to true (line 1). If a result set S_r is not empty, for each $r_i = \langle c, dist_{pt}, price \rangle$ in S_r, if $ldist(g_j, s) > max\{r_i.dist_{pt}, r_i.price/f_n - 2 \times dist(s, d)\}$, the algorithm sets *flag* to false and breaks, i.e., the empty-vehicle list $g_j.L_e$ of g_j can be pruned safely according Lemma 2 (lines 2-5). After that, if *flag* is true, the algorithm verifies the empty vehicles (lines 6-12). For each vehicle $c_k \in g_j.L_e$ that is not verified, it sets *dominate* to true (line 8), and verifies c_k using all the results in S_r (lines 9-11). For each $r_i = \langle c, dist_{pt}, price \rangle$ in S_r, due to Lemma 1, if $ldist(c_k.l, s) > max\{r_i.dist_{pt}, r_i.price/f_n - 2 \times dist(s, d)\}$, then the algorithm sets *dominate* to false and breaks (lines 10-11). After all verifications, if *dominate* is true, c_k is added to S_c (line 12). Finally, S_c is returned (line 13).

Algorithm 3 gives the pseudo-code of *find_nonempty_vehicle*. It first initializes a vehicle candidate set S_c as empty, and sets *flag* and *dominate* to true (line 1). If $max\{o_x.capacity \mid \langle o_x, o_y \rangle \in g_j\} < n$ or $max\{o_x.detour \mid \langle o_x, o_y \rangle \in g_j\} < 2 \times ldist(s, g_j) - max\{dist(o_x, o_y) \mid \langle o_x, o_y \rangle \in g_j\}$, it returns the candidate vehicle set S_c due to Lemma 6 (lines 2-3). When the result set S_r is not empty, for each $r_i = \langle c, dist_{pt}, price \rangle$ in S_r, if $ldist(s, g_j) + min\{o_x.dist_{tr} \mid \langle o_x, o_y \rangle \in g_j\} > r_i.dist_{pt}$ and $2 \times ldist(s, g_j) - max\{dist(o_x, o_y) \mid \langle o_x, o_y \rangle \in g_j\} > r_i.price/f_n - dist(s, d)$, it sets *flag* to false and breaks, i.e., all the kinetic tree edges $\langle o_x, o_y \rangle$ contained in $g_j.L_n$ can be pruned by Lemma 4 (lines 4-7). Next, the algorithm verifies the non-empty vehicles if *flag* is true (lines 8-16). The processing is similar to Algorithm 2, the only difference is that it utilizes Lemmas 3 and 5 instead of Lemma 1 to avoid unnecessary verifications of non-empty vehicles. Finally, it returns the candidate vehicle set S_c.

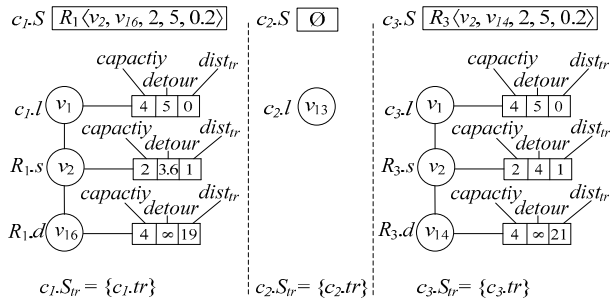
Algorithm 4 Find_Result Algorithm

Input: the vehicle c, the result set S_r
1: **if** c is empty **then**
2: insert $\langle c, dist(c.l, s), f_n \times (dist(c.l, s) + 2 \times dist(s, d)) \rangle$ into S_r
3: **else**
4: $S_r = S_r \cup insert\_request(R, c.Str)$ // integrate Lemmas 3, 5, 7, and 11

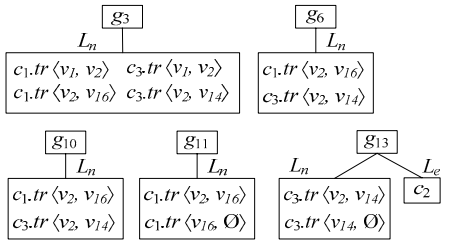
Algorithm 4 gives the pseudo-code of *find_result*. It inserts $\langle c, dist(c.l, s), f_n \times (dist(c.l, s) + 2 \times dist(s, d)) \rangle$ into S_r for an empty vehicle c (lines 1-2). For a non-empty vehicle c, the algorithm involves *insert_request* to insert the request R into c's kinetic tree (similar to an existing the kinetic tree insertion method [17] by integrating Lemmas 3, 5, 7, and 11), which returns a set of qualified results. Thus, $S_r = S_r \cup insert\_request(R, c.Str)$ (lines 3-4). Note that, when inserting the new result r into S_r, existing results dominated by r can be pruned.

Example 1. We illustrate SSA using the example in Fig. 6, where each of the two non-empty vehicles c_1 and c_3 are assigned a request, and c_2 is an empty vehicle. Assume that a new ridesharing request $R_2 = \langle v_{12}, v_{17}, 2, 5, 0.2 \rangle$. First, SSA initializes the result set $S_r = \emptyset$ and a vehicle candidate set $S_c = \emptyset$. It then obtains a grid cell set $S_g = \{g_{10}, g_6, g_{11}, g_{13}, g_3\}$. Then, the algorithm verifies the grid cells in S_g in order.

For grid cell g_{10}, as its L_e is empty, find_empty_vechile returns an empty set. Since $S_r = \emptyset$ and $S_c = \emptyset$, L_n of g_{10} cannot be pruned, and the non-empty vehicles c_1 and c_3 in L_n also cannot be pruned. Hence, $S_c = \{c_1, c_3\}$. For vehicle c_1 in S_c, the algorithm inserts R_2 into its kinetic tree, resulting in a new valid trip schedule $\langle v_1, v_2, v_{12}, v_{16}, v_{17} \rangle$ with the corresponding result $r_1 = \langle c_1, 15, 4 \rangle$. During the insertion, (i) the start location v_{12} of R_2 cannot be inserted into $\langle v_1, v_2 \rangle$ based on



(a) The trip schedules of all vehicles



(b) The empty and non-empty vehicles in the grid index of Fig. 2

Fig. 6. Example for SSA and DSA

Lemma 5, and it cannot be inserted into $\langle v_{16}, \emptyset\rangle$ *due to Lemma 3*; (*ii*) *the result* $r_1' = \langle c_1, 15, 4.4\rangle$ *of another valid trip schedule* $\langle v_1, v_2, v_{12}, v_{17}, v_{16}\rangle$ *is dominated by* r_1. *For vehicle* c_3 *in* S_c, *the start location* v_{12} *of* R_2 *cannot be inserted into* $\langle v_1, v_2\rangle$, $\langle v_2, v_{14}\rangle$, $\langle v_{14}, \emptyset\rangle$ *due to Lemma 3. Thus, no valid result is returned by* c_3. *After processing the grid cell* g_{10}, $S_r = \{\langle c_1, 15, 4\rangle\}$. *Then SSA processes the grid cells* g_6, g_{11}, g_{13}, *and* g_3 *similarly. Finally,* $S_r = \{\langle c_1, 15, 4\rangle, \langle c_2, 8, 8.8\rangle\}$ *is returned.*

Discussion. In the worst case, SSA must traverse the grid index and all the kinetic trees once. However, compared with the Baseline Algorithm (BA), SSA avoids unnecessary vehicle verifications and shortest path distance computations. In *Example 1*, although SSA needs to verify all vehicles like BA in this example, it does not need to compute any shortest path distance when verifying vehicle c_3, because the start location v_{12} of R_2 cannot be inserted into $\langle v_1, v_2\rangle$, $\langle v_2, v_{14}\rangle$, $\langle v_{14}, \emptyset\rangle$ due to Lemma 3. Hence, SSA is more efficient than Baseline. Note that, the space cost of SSA equals to the size of indexes, as discussed in Section IV.

C. Dual-Side Search Algorithm

The single-side search algorithm filters unqualified vehicles using pruning heuristics when inserting the start location. Similarly, we can filter unqualified vehicles using pruning heuristics when inserting the destination location. Take the following case as an example. One existing trip schedule is near the start location of a ridesharing request, but it is far from the destination of the ridesharing request. In this example, the dual-side search can further avoid unnecessary verifications of vehicles. Specifically, we utilize Lemmas 3~6 to filter unqualified non-empty vehicles when inserting the start location, and also use Lemmas 7~10 to filter unqualified non-empty vehicles when inserting the destination. After that, as does SSA, we verify each vehicle that cannot be pruned to find qualified results. Here, we only verify the non-empty vehicles not pruned by the start or destination location.

Algorithm 5 Dual-Side Search Algorithm (DSA)

Input: a ride request $R = \langle s, d, n, w, \varepsilon\rangle$, a grid index on a road network G
Output: the result set S_r for R
/* S_s: the candidate vehicle set found from a start location s
S_d: the candidate vehicle set found from a destination location d */
1: $S_r = \emptyset$, $S_s = \emptyset$, $S_d = \emptyset$, $S_g = \{g_i\} \cup g_i.L_g$ and $S_h = \{g_j\} \cup g_j.L_g$
// s locates in the grid cell g_i and d locates in the grid cell g_j
2: $g_s = S_g.first()$, $g_d = S_h.first()$ // get the first element
3: **while** $g_s \neq \emptyset$ and $g_d \neq \emptyset$ **do**
4: $S_1 = find\_empty\_vehicle(g_s, S_r)$ // Algorithm 2
5: $S_2 = find\_nonempty\_vehicle(g_s, S_r)$ // Algorithm 3
6: $S_3 = find\_nonempty\_vehicle\_Dest(g_d, S_r)$
7: $S_s = S_s \cup S_2$, $S_d = S_d \cup S_3$
8: $I = S_1 \cup (S_s \cap S_3) \cup (S_d \cap S_2)$
9: **for** each vehicle c *in* I **do**
10: *insert_request*(R, c, S_r) // Algorithm 4
11: $g_s = S_g.next()$, $g_d = S_h.next()$ // get the next element
12: **return** S_r

Function: *find_nonempty_vehicle_Dest*(g_d, S_r)
13: $S_c = \emptyset$, *flag* = true, *dominate* = true
14: **if** $max\{o_x.capacity \mid \langle o_x, o_y\rangle \in g_d\} < n$ or $max\{o_x.detour \mid \langle o_x, o_y\rangle \in g_d\} < 2 \times ldist(d, g_d) - max\{dist(o_x, o_y) \mid \langle o_x, o_y\rangle \in g_d\}$ **then**
15: **return** S_c // Lemma 8
16: **for** each $r_i = \langle c, dist_{pt}, price\rangle$ in S_r **do**
17: **if** $min\{o_x.dist_{tr} \mid \langle o_x, o_y\rangle \in g_d\} + ldist(g_j, d) - (1+\varepsilon) \times dist(s, d) > r_i.dist_{pt} \wedge 2 \times ldist(d, g_d) - max\{dist(o_x, o_y) \mid \langle o_x, o_y\rangle \in g_d\} > r_i.price/f_n - dist(s, d)$ **then**
18: *flag* = false and **break** // Lemma 10
19: **if** *flag* **then**
20: **for** each $\langle o_x, o_y\rangle$ of $c_k.tr \in g_d.L_n \wedge c_k$ is not verified **do**
21: *dominate* = true
22: **if** $o_x.capacity < n$ or $o_x.detour < ldist(d, o_x) + ldist(d, o_y) - dist(o_x, o_y)$ **then**
23: *dominate* = false and **break** // Lemma 7
24: **for** each $r_i = \langle c, dist_{pt}, price\rangle$ in S_r **do**
25: **if** $dist_{tr}(c_k.l, o_x) + ldist(o_x, d) - (1 + \varepsilon) \times dist(s, d) > r_i.dist_{pt} \wedge ldist(o_x, d) + ldist(o_y, d) - dist(o_x, o_y) > r_i.price/f_n - dist(s, d)$ **then**
26: *dominate* = false and **break** // Lemma 9
27: **if** *dominate* **then** insert c_k into S_c
28: **return** S_c

Algorithm 5 presents the pseudo-code of Dual-Side Search Algorithm (DSA). It takes as inputs a ridesharing request $R = \langle s, d, n, w, \varepsilon\rangle$ and a grid index on the road network, and it outputs the result set S_r for R. First, DSA initializes a result set S_r, a candidate vehicle set S_s found from the start location, and the candidate vehicle set S_d found from the destination as empty. In addition, it sets the grid cell set S_g to $\{g_i\} \cup g_i.L_g$ and the grid cell set S_h to $\{g_j\} \cup g_j.L_g$ (line 1). Here, g_i and g_j are the current grid cells that contain the start and destination locations of R locate respectively, and $g_i.L_g$ and $g_j.L_g$ contain the other grid cells sorted in ascending order of their minimal distances to g_i and g_j, respectively. Then, DSA sets g_s and g_d to the first element of S_g and S_h, respectively (line 2). Next, a while loop is performed until g_s and g_d are empty (lines 3-11). In each while loop, DSA calls *find_empty_vehicle* to find the candidate empty vehicle set S_1 in g_s (line 4), invokes *find_nonempty_vehicle* to find the candidate nonempty vehicle set S_2 using the start location (line 5), and uses *find_nonempty_vehicle_Dest* to find the candidate vehicle set S_3 using the destination location (line 6). Then, it computes $S_s = S_s \cup S_2$, $S_d = S_d \cup S_3$, and $I = S_1 \cup (S_s \cap S_3) \cup (S_d \cap S_2)$ to find candidate vehicle set I (lines 7-8). Thereafter, for each

candidate vehicle c in I, it invokes *find_result* to find the final results (lines 9-10). In the sequel, g_s and g_d are set to the next elements of S_g and S_h, respectively (line 11). Finally, DSA returns the result set S_r (line 12).

Function *find_candidate_vehicle_Dest* first initializes the vehicle candidate set S_c as empty, and sets *flag* and *dominate* to true (line 13). If $max\{o_x.capacity \mid \langle o_x, o_y\rangle \in g_d\} < n$ or $max\{o_x.detour \mid \langle o_x, o_y\rangle \in g_d\} < 2 \times ldist(d, g_d) - max\{dist(o_x, o_y) \mid \langle o_x, o_y\rangle \in g_d\}$, it returns S_c, i.e., all kinetic tree edges $\langle o_x, o_y\rangle$ contained in $g_d.L_n$ can be pruned according to Lemma 8 (lines 14-15). For each result $r_i = \langle c, dist_{pt}, price\rangle$ in S_r, if $min\{o_x.dist_{tr} \mid \langle o_x, o_y\rangle \in g_d\} + ldist(g_d, d) - (1 + \varepsilon) \times dist(s, d) > r_i.dist_{pt}$ and $2 \times ldist(d, g_d) - max\{dist(o_x, o_y) \mid \langle o_x, o_y\rangle \in g_d\} > r_i.price/f_n - dist(s, d)$, it sets *flag* to false and breaks due to Lemma 10 (lines 16-18). After that, the function verifies the non-empty vehicles if *flag* is true (lines 19-27). For each vehicle $c_k \in g_d.L_n$ that is not verified, it sets *dominate* to true (line 21); and if $o_x.capacity < n$ or $o_x.detour < ldist(d, o_x) + ldist(d, o_y) - dist(o_x, o_y)$, it sets *dominate* to false and breaks by Lemma 7 (lines 22-23). Next, it verifies c_k using all the results in S_r (lines 24-26). For each result $r_i = \langle c, dist_{pt}, price\rangle$ in S_r, if $dist_{tr}(c_k.l, o_x) + ldist(o_x, d) - (1 + \varepsilon) \times dist(s, d) > r_i.dist_{pt}$ and $ldist(o_x, d) + ldist(o_y, d) - dist(o_x, o_y) > r_i.price/f_n - dist(s, d)$, the function sets *dominate* to false and breaks based on Lemma 9 (lines 25-26). After the verification, if *dominate* is true, then c_k is inserted to S_c (line 27). Finally, it returns the candidate vehicle set S_c (line 28).

Example 2. We illustrate DSA using the same example in Fig. 6. Consider a new ridesharing request $R_2 = \langle v_{12}, v_{17}, 2, 5, 0.2\rangle$. First, DSA initializes $S_r = \emptyset$, $S_s = \emptyset$, and $S_d = \emptyset$, and then sets $S_g = \{g_{10}, g_6, g_{11}, g_{13}, g_3\}$ and $S_h = \{g_{11}, g_{10}, g_6, g_3, g_{13}\}$. Then, the algorithm processes the grid cells extended from start and destination locations simultaneously.

For $g_s = g_{10}$ and $g_d = g_{11}$, DSA finds the candidate empty vehicles $S_1 = \emptyset$, as L_e is empty; finds the candidate non-empty vehicles $S_2 = \{c_1, c_3\}$ using the start location the same as that in Example 1; and finds the candidate non-empty vehicles $S_3 = \{c_1\}$ using the destination location as S_r is empty. After that, we can get that $S_s = \{c_1, c_3\}$, $S_d = \{c_1\}$, *and* $I = \{c_1\}$. *For c_1 in I, the algorithm inserts R_2 into its kinetic tree, and returns the result $r_1 = \langle c_1, 15, 4\rangle$ as in Example 1. Then, DSA processes $\langle g_6, g_{10}\rangle$, $\langle g_{11}, g_6\rangle$, and $\langle g_{13}, g_3\rangle$ similarly. Finally, $S_r = \{\langle c_1, 15, 4\rangle, \langle c_2, 8, 8.8\rangle\}$ is returned.*

Discussion. In the worst case, DSA traverses the grid index twice and traverses all the kinetic trees once, which is more costly than SSA. However, compared with SSA, DSA further reduces the number of verified vehicle. As shown in *Example 2*, DSA only needs to verify vehicles c_1 and c_2, while SSA needs to verify c_1, c_2, and c_3. In general, the performance of DSA depends on the request distribution. In other words, if the DSA can avoid lots of vehicle verifications, it can achieve better performance than SSA. Similar as SSA, DSA performs on the indexes, and thus, the space cost of DSA equals to the size of indexes. To further improve the performance of DSA, we only search part of the grid for both SSA and DSA, as to be verified in Section VII.A.

TABLE II
PARAMETER SETTINGS

Parameter	Setting	Default
waiting time w (minute)	2, 3, 4, 5, 6	2
service constraint ε	0.2, 0.3, 0.4, 0.5, 0.6	0.2
capacity of vehicle	2, 3, 4, 5, 6	4
the number of vehicles	12K, 14K, 16K, 18K, 20K	20K
the grid cell size (meter)	3333, 2000, 1429, 1111, 909	2000
the number of requests	1000, 3000, 5000, 7000, 9000	1000
the number of verified grids (%)	8, 16, 32, 64, 100	16

VII. EXPERIMENTS

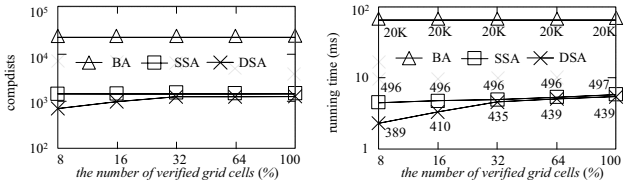
In this section, we experimentally evaluate all three algorithms using a large taxi dataset containing 432,327 trips made by 17,000 Shanghai taxis over one day (May 29, 2009) [17]. A trip tp has the start time $tp.st$, the start location $tp.sl$, the end time $tp.et$, and the end location $tp.el$. Hence, a simulator generates ridesharing requests by using trips from the dataset during a certain time period. Specifically, for each trip tp, a ridesharing request R is initialized as $R = \langle tp.sl, tp.el, n, w, \varepsilon\rangle$, and R is submitted at time $tp.st$, where n, w and ε are fixed. Although the submission time does not equal to the start time, we use the start time to simulate the submission time. In the taxi dataset, multiple requests may occur every second. When multiple requests occur, a greedy strategy is used as discussed in Section III.E.

The Shanghai road network is represented by an undirected and weighted graph containing 122,319 vertices and 188,426 edges. The road network and the corresponding index are stored in main memory. The start and destination locations of each request are pre-mapped to the closest vertex in the graph. A vehicle is also initialized to a random vertex in the road network, which follows the uniform distribution in the grid. Vehicles follow a given route when customers are on board or, otherwise, follow the current road segment, choosing a random segment to follow at intersections. A constant speed D is assumed, specifically, we set D to 48km/hr. Time-distance conversion is accomplished by multiplying or dividing by D.

We explore the performance using the settings shown in Table II. In every experiment, we change one parameter, and set the others to their default values. The main performance metric includes the running time, the number of shortest path distance computations (*compdists* for short), and the number of verified vehicles. Every measurement reported is an average across all the ridesharing requests. All the algorithms are implemented in C++, and run on a PC with an Intel Core i7 3.6GHz CPU and a 32GB RAM.

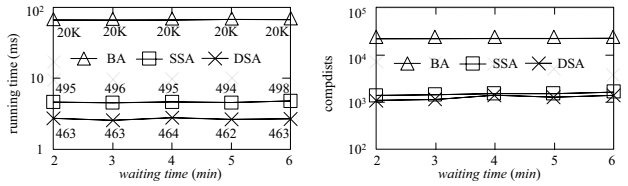
A. Effect of the Number of Verified Grid Cells

We first study the effect of the number of verified grid cells on the efficiency of Baseline Algorithm (BA), the Single-side Search Algorithm (SSA), and the Dual-side Search Algorithm (DSA). Fig. 7 shows the results. The number of verified vehicles of each algorithm is shown as an integer together with running time. As discussed in VI.C, the cost of DSA increases rapidly (i.e., its efficiency degrades), while the costs of SSA and BA increase slightly with the number of verified grid cells. Table III lists the precision and recall. As observed, SSA and DSA achieve high result quality by searching part of the grid.



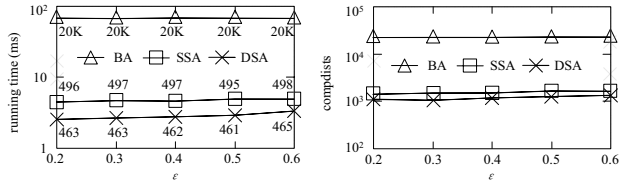
(a) Running time and number of verified vehicles (b) *compdists*

Fig. 7. Performance cost vs. the number of verified grid cells



(a) Running time and number of verified vehicles (b) *compdists*

Fig. 8. Performance cost vs. waiting time w



(a) Running time and number of verified vehicles (b) *compdists*

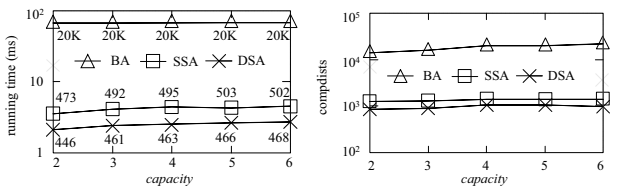
Fig. 9. Performance cost vs. service constraint ε

B. Effect of Waiting Time

Fig. 8 depicts the experimental results for various waiting times. The first observation is that SSA and DSA perform better than BA, while DSA performs the best. Because SSA and DSA use techniques to avoid unnecessary vehicle verifications and shortest path distance computations. In addition, DSA filters the vehicles using the dual-side search paradigm while SSA uses single-side search paradigm. The dual-side search paradigm can avoid verifying vehicles whose valid trip schedule is near the start location, but far from the destination location. Hence, DSA performs the best. The second observation is that, the costs increase slightly with the growth of the waiting time at most of times. This is because, as the waiting time increases, more vehicles satisfy the waiting time constraint, resulting in more vehicles to be verified, in turn increasing the number of *compdists* to be computed and the running time to be consumed. However, the cost (e.g., the number of verified vehicles) could drop. This is because, when the waiting time increases (i.e., the constraint becomes loser), it is possible to answer a new request with a better result that can prune unnecessary verifications.

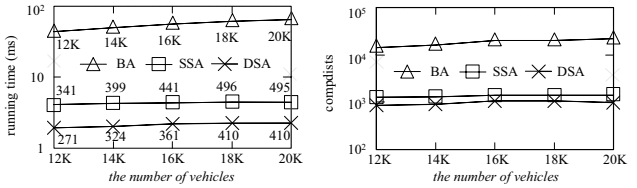
C. Effect of Service Constraint

Fig. 9 shows the experimental results when varying ε from 0.2 to 0.6. We again observe that SSA and DSA are much more efficient than BA, while DSA performs the best. Second, the costs increase with the growth of ε. The reason is that as ε increases, more vehicles satisfy the service constraint, resulting in more verified vehicles, more *compdists*, and longer running time.



(a) Running time and number of verified vehicles (b) *compdists*

Fig. 10. Performance cost vs. capacity of vehicle



(a) Running time and number of verified vehicles (b) *compdists*

Fig. 11. Performance cost vs. the number of vehicles

TABLE III
PRECISION AND RECALL VS. THE NUMBER OF VERIFIED GRIDS

	8%	**16%**	**32%**	**64%**	**100%**
SSA	1/1	1/1	1/1	1/1	1/1
DSA	0.94/1	0.96/1	0.998/1	0.999/1	1/1

D. Effect of Capacity of Vehicle

Fig. 10 depicts the experimental results when varying the capacity of a vehicle from 2 to 6. As observed, the costs increase with the growth of capacity. This is because, when the capacity is small, the capacity constraint eliminates more vehicles. In an extreme case, when the capacity equals to 2, only two riders can share, and thus, many vehicles can be filtered for a new request. When the capacity increases, the capacity constraint becomes loser, and thus, the costs increase.

E. Effect of Number of Vehicles

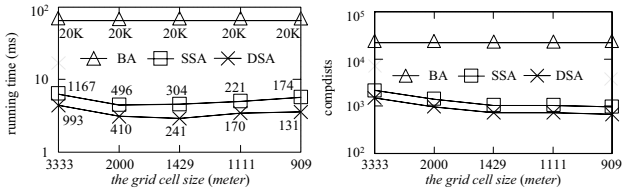
Fig. 11 illustrates the experimental results under various numbers of vehicles. We can observe that the costs increase with the growth of number of vehicles due to the larger search space. In addition, we can observe that the costs of SSA and DSA increase more slowly than that of BA. This is because, the search space of SSA and DSA is smaller than that of BA, i.e., BA needs to verify all the vehicles, while SSA and DSA only verify the vehicles near the start and destination locations.

F. Effect of Grid Cell Size

Fig. 12 shows the experimental results when varying the grid cell size. As observed, *compdists* and the number of verified vehicles drop with the grid cell size, due to better pruning power. However, the running time first decreases and then increases, as the pruning cost grows. In addition, Table IV depicts the maximal memory costs for the grid index and the kinetic trees, while the fixed memory cost for the road network is 67M. As expected, the memory cost of the grid index grows rapidly as the grid cell size drops.

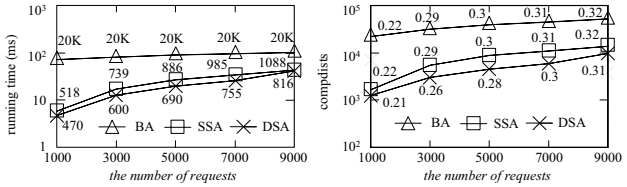
G. Effect of the Number of Requests

Finally, to verify the scalability of our presented algorithms, we vary the number of requests from 1000 to 9000. Fig. 13 plots the experimental results, where the fractions in Fig. 13(b)



(a) Running time and number of verified vehicles (b) *compdists*

Fig. 12. Performance cost vs. the grid cell size



(a) Running time and number of verified vehicles (b) *compdists*

Fig. 13. Performance cost vs. the number of requests

denote the sharing rates (i.e., the fractions of the requests that share vehicles). It is observed that, the costs increase with the growth of the number of requests. The reason is that, the sharing rate increases when the number of requests grows.

VIII. CONCLUSIONS

Ridesharing offers economic rewards to travelers and holds the potential to contribute to reducing the environmental impact of transportation. To enable more attractive ridesharing, we consider both the pick-up time and price of ridesharing options and develop a solution that presents all non-dominated options in response to a ridesharing request. To accelerate the matching between requests and vehicles, indexes and corresponding pruning heuristics are developed. We also propose algorithms that follow the single-side and dual-side search paradigms in order to reduce the number of vehicle verifications and shortest path distance computations needed. Extensive experiments on a large Shanghai taxi data set offer evidence that our single-side and dual-side search algorithms are much faster than the state-of-the-art method, and then show that the dual-side search algorithm performs the best.

In the future, we plan to extend our ridesharing system to a distributed platform (e.g., Flink or Spark). Another promising direction is to investigate more efficient indexes for dynamic ridesharing with fast update operations.

ACKNOWLEDGEMENTS

This work was supported in part by the 973 Program Grant No. 2015CB352502, NSFC Grant No. 61522208 and 61379033, NSFC-Zhejiang Joint Fund No. U1609217, grant MOE2015-T2-2-069 from MOE Singapore, grant SUG from NUS, the DiCyPS project, and the Obel Family Foundation.

TABLE IV
MEMORY COST VS. THE GRID CELL SIZE

	3333m	**2667m**	**2000m**	**1600m**	**1333m**
grid index	100M	764M	2933M	8012M	17877M
kinetic trees	2.19M	2.21M	2.21M	2.24M	2.24M

REFERENCES

[1] [online] lyft. https://www.lytf.com.
[2] [online] uberPOOL. https://www.uber.com.
[3] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang, "Optimization for dynamic ride-sharing: A review," *EJOR*, vol. 223, pp. 295–302, 2012.
[4] M. Asghari, D. Deng, C. Shahabi, U. Demirurek, and Y. Li, "Price-aware real-time ride-sharing at scale: An auction-based approach," in *SIGSPATIAL GIS*, article 3, 2016.
[5] G. Berbeglia, J. F. Cordeau, and G. Laporte, "Dynamic pickup and delivery problems," *European Journal of Operational Research*, vol. 202, no. 1, pp. 8–15, 2010.
[6] S. Borzsonyi, D. Kossmann, and L. Stocker, "The skyline operator," in *ICDE*, pp. 421–430, 2001.
[7] B. Cao, L. Alarabi, M. F. Mokbel, and A. Basalamah, "SHAREK: A scalable dynamic ride sharing system," in *MDM*, pp. 4–13, 2015.
[8] P. Cheng, H. Xin, and L. Chen, "Utility-aware ridesharing on road networks," in *SIGMOD*, pp. 1197–1210, 2017.
[9] C. Dai, "Ridesharing recommendation: Whether and Where should I wait?" in *WAIM*, pp. 151–163, 2006.
[10] X. Duan, C. Jin, X. Wang, A. Zhou, and K. Yue, "Real-time personalized taxi-sharing," in. *DASFAA*, pp. 451–465, 2016.
[11] M. Furuhata, M. Dessouky, F. Ordonez, M.-E. Brunet, X. Wang, and S. Koenig, "Ridesharing: The state-of-the-art and future directions," *Transportation Research Part B: Methodological*, vol. 57, pp. 28–46, 2013.
[12] G. Gidofalvi, T. B. Pedersen, T. Risch, and E. Zeitler, "Highly scalable trip grouping for large-scale collective transportation systems," in *EDBT*, pp. 678–689, 2008.
[13] J. Hu, B. Yang, C. S. Jensen, and Y. Ma, "Enabling time-dependent uncertain eco-weights for road networks," *GeoInformatica*, vol. 21, no. 1, pp. 57–88, 2017.
[14] S. Ma and O. Wolfson, "Analysis and evaluation of the slugging from of ridesharing," in *SIGSPATIAL GIS*, pp. 64–73, 2013.
[15] S. Ma, Y. Zheng, and O. Wolfson, "T-share: A large-scale dynamic taxi ridesharing service," *in ICDE*, pp. 410–421, 2013.
[16] M. E. T. Horn, "Fleet scheduling and dispatching for demand-responsive passenger services," *Transp. Res. Part C Emerg. Technol.*, vol. 10, no. 1, pp. 35–63, 2002.
[17] 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.
[18] D. H. Lee, H. Wang, R. L. Cheu, and S. H. Teo, "Taxi dispatch system based on current demands and real-time traffic conditions," *in Transportation Network Modelling*, pp. 193–200, 2004.
[19] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Optimal multi-meeting-point route search," *IEEE TKDE*, vol. 28, no. 3, pp. 770–784, 2016.
[20] P. M. d'Orey, R. Fernandes, and M. Ferreira, "Empirical evaluation of a dynamic and distributed taxi-sharing system," in *ITSC*, pp. 140–146, 2012.
[21] W. B. Powell, B. Bouzanene-Ayari, and H. P. Simpo, "Dynamic models for freight transportation," *Handbooks in Operations Research and Management Science*, vol. 14, pp. 285–365, 2007.
[22] A. B. T. Sherif, K. Rabieh, M. M. E. A. Mahmoud, and X. Liang, "Privacy-preserving ridesharing scheme for autonomous vechicles in big data era," *IEEE Internet Things J.*, vol. 4, no. 2, pp. 611–618, 2017.
[23] L. Song, C. Wang, X. Duan, B. Xiao, X. Liu, R. Zhang, X. He, and X. Gong, "TaxiHailer: A situation-specific taxi pick-up points recommendation system," in *DASFAA*, pp. 523–526, 2014.
[24] R. S. Thangaraj, K. Mukherjee, G. Raravi, A. Metewar, "Xhare-a-Ride: A search optimized dynamic ride sharing system with approximation guarantee," in *ICDE*, pp. 1117–1128, 2017.
[25] W. Tong, J. Hua, and S. Zhong, "A jointly differentially private scheduling protocol for ridesharing services," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 10, pp. 2444–2456, 2017.
[26] K. Wong, I. Bell, and G. H. Michael, "Solution of the dial-a-ride problem with multi-dimensional capacity constraint," *Int. Trans. Oper. Res.*, vol. 13, no. 3, pp. 195–208, 2006.
[27] S. Yeung, E. Miller, and S. Madria, "A flexible real-time ridesharing system considering current road conditions," in *MDM*, pp. 186–191, 2016.
[28] J. Zhang, D. Wen, and S. Zeng, "A discounted trade reduction mechanism for dynamic ridesharing pricing," *IEEE Transactions on Intelligent Transportation systems*, vol. 17, no. 6, pp. 1586–1595, 2016.