T-Share: A Large-Scale Dynamic Taxi Ridesharing Service

Shuo Ma[1,2*], Yu Zheng[2], Ouri Wolfson[1,2]
[1]*University of Illinois at Chicago, Chicago, USA*
{sma, wolfson}@cs.uic.edu
[2]*Microsoft Research Asia, Beijing, China*
yuzheng@microsoft.com

Abstract— **Taxi ridesharing can be of significant social and environmental benefit, e.g. by saving energy consumption and satisfying more people's commute needs in peak hours. Despite the great potential, taxi ridesharing, especially with dynamic queries, is not well studied. In this paper, we formally define the dynamic ridesharing problem and propose a large-scale taxi ridesharing service. It efficiently serves real-time requests sent by taxi users and generates ridesharing schedules that reduce the total travel distance significantly. In our method, we first propose a taxi searching algorithm using a spatio-temporal index to quickly retrieve candidate taxis that are likely to satisfy a user query. A scheduling algorithm is then proposed. It checks each candidate taxi and inserts the query's trip into the schedule of the taxi which satisfies the query with minimum additional incurred travel distance. To tackle the heavy computational load, a lazy shortest path calculation strategy is devised to speed up the scheduling algorithm. We evaluated our service using a GPS trajectory dataset generated by over 33,000 taxis during a period of 3 months. By learning the spatio-temporal distributions and the stochastic process of real user queries from this dataset, we built an experimental platform that can simulate user behaviours in taking a taxi in the real-world. Tested on this platform with extensive experiments, our approach demonstrated its efficiency, effectiveness, and scalability. For example, our proposed service can serve 25% additional taxi users while saving 13% travel distance compared with no-ridesharing (when the ratio of the number of queries to the number of taxis is 6).**

I. INTRODUCTION

Ridesharing is a promising approach for saving energy consumption and assuaging traffic congestion while satisfying people's needs in commute. Ridesharing based on private cars, often known as carpooling or recurring ridesharing, has been studied for years to deal with people's routine commutes, e.g., from home to work [6][2]. Recently it became more and more difficult for people to hail a taxi during rush hours in increasingly crowded urban areas. Naturally, taxi ridesharing [12] is considered as a potential approach to tackle this emerging transportation headache.

In contrast to the recurring ridesharing, taxi ridesharing is more challenging as both passengers' queries and positions of taxis are highly dynamic and difficult to predict: 1) any user can submit a query anytime and anywhere, which is real-time in most cases; and 2) a taxi constantly travels on roads, picking passengers up and dropping them off.
Its destination depends on that of passengers, while passengers could go anywhere in a city.

[*] The work was done when the author was doing an internship in Microsoft Research Asia under the supervision of the second author.

In this paper, we study the dynamic taxi ridesharing problem in a practical setting and design a system called *T-Share*. Consider that an organization (e.g. a company or a transportation authority) that operates a dynamic taxi ridesharing service. Taxi drivers can independently determine when to join and leave the service. Passengers submit ride queries in real time via a mobile device, e.g., a smart phone (assume they are willing to share the ride with others). Each query indicates the origin and destination locations of the trip, as well as time windows constraining when the passenger wants to be picked up and dropped off. On receiving a new query, the operation centre will dispatch an "appropriate" taxi which is able to satisfy both the new query and the trips of existing passengers who are already assigned to the taxi. The updated schedules and routes will be then given to the corresponding taxi driver and passengers.

In this study, we pursue a two-fold goal. The primary purpose is to investigate the potential of taxi ridesharing in serving more taxi ride queries by comparing with the case where no ridesharing is conducted. Additionally, we try to reduce the total travel distance of these taxis (so as to reduce the energy consumption) when doing the ridesharing. The second goal is to build a dynamic ridesharing service applicable for the practical use, i.e. serving a large number of queries quickly. We will show how we approach this goal by applying a fast taxi searching algorithm and the lazy shortest path calculation strategy.

To the best of our knowledge, our work is the first to consider dynamic ridesharing for a large number of taxis. We place our problem in a practical setting by exploiting a real city road network and the enormous historical taxi trajectory data. The contribution of this paper is multiple-dimensional:

- We propose a taxi searching algorithm and a scheduling algorithm which together can quickly serve dynamic queries while significantly reducing the total travel distance of taxis in ridesharing. That is, these algorithms promise a small query processing time; and at the same time a large amount of energy and greenhouse gas emission is saved.
- By exploiting the taxi trajectory dataset, we build an experimental platform which can produce taxi ride queries conforming to the real query distribution over space and time. We believe that the platform is of great value in many other related urban and transportation computation problems such as traffic prediction.
- We perform extensive experiments to validate the effectiveness of taxi ridesharing as well as the efficiency

and scalability of our proposed taxi ridesharing service. According to the experimental results, the fraction of queries that get satisfied is increased by 25% via ridesharing when taxis are in high demand. Furthermore, 120 million liter of gasoline can be saved each year in Beijing by taxis alone if ridesharing is allowed.

The rest of this paper is organized as follows. In Section II, we formally define the dynamic taxi ridesharing problem and overview our proposed service. Section III introduces the spatio-temporal index of taxis. Section IV describes two flavours of taxi searching algorithms. Section V describes the scheduling module and the lazy shortest path calculation. We present the evaluation in Section VI and summarize the related work in Section VII.

II. OVERVIEW

A. Preliminaries

Definition 1 (Query) A *query* Q is a passenger's request for a taxi ride that is associated with a timestamp $Q.t$ indicating when the query is submitted, a pickup point $Q.o$, a delivery point $Q.d$, a time window $Q.wp$ defining the time interval when the passenger needs to be picked up at the pickup point, and a time window $Q.wd$ defining the time interval when the passenger needs to be dropped off at the delivery point. The early and late bounds of a pickup window are denoted by $Q.wp.e$ and $Q.wp.l$; Likewise, $Q.wd.e$ and $Q.wd.l$ stand for that of a delivery window.

For the sake of description simplicity, each query here only represents one passenger. But our approach is readily applied to the case where a query represents multiple passengers. In practice, a passenger only needs to explicitly indicate $Q.d$ and $Q.wd.l$, as most information of a query can be automatically obtained from a passenger's mobile phone, e.g., $Q.o$ and $Q.t$. In addition, we assume that both $Q.wp.e$ and $Q.wd.e$ equals to $Q.t$, and $Q.wp.l$ can be easily obtained by adding a fixed value, e.g. 5 minutes, to $Q.wp.e$.

Definition 2 (Schedule) A *schedule* s is a temporally-ordered sequence of pickup and delivery points of n queries $Q_1, Q_2, \ldots\ldots Q_n$ such that for every query Q_i, i=1,…n, either 1) $Q_i.o$ precedes $Q_i.d$ in the sequence, or 2) only $Q_i.d$ exists in the sequence.

It is clear from the definition that the schedule dynamically changes over time. For example, a schedule involving 2 queries Q_1 and Q_2 could be $Q_1.o \rightarrow Q_2.o \rightarrow Q_1.d \rightarrow Q_2.d$ at a certain time. The schedule is updated to $Q_2.o \rightarrow Q_1.d \rightarrow Q_2.d$ once the taxi has passed pickup point $Q_1.o$.

Definition 3 (Taxi Status) A *taxi status* V represents the instantaneous state of a taxi and is comprised of a taxi identifier $V.id$, a timestamp $V.t$, a geographical location $V.l$, the number of on-board passengers $V.p$ and a schedule $V.s$.

Definition 4 (Satisfaction) Given a taxi status V and a query Q, we say that V *satisfies* Q if and only if (i) $V.p$ is smaller than the seat capacity of the taxi; (ii) V can pick up the passenger of Q at $Q.o$ no later than $Q.wp.l$, and drop off her at $Q.d$ no later than $Q.wd.l$; (iii) V can pick up and drop off existing passengers in $V.s$ no later than the late bound of their corresponding pickup and delivery time windows.

TABLE I summarizes a list of essential notations used throughout the paper. (Some notations are introduced later.)

TABLE I A LIST OF NOTATIONS

Notation	Definition
Q	A query for a taxi ride
$Q.t$	The birth time of query Q
$Q.o$	The pickup point of query Q
$Q.d$	The delivery point of query Q
$Q.wp$	The pickup time window of query Q
$Q.wd$	The delivery time window of query Q
V	A taxi status
$V.s$	The current schedule of taxi status V
$V.l$	The current location of taxi status V
g_i	A grid cell
c_i	The anchor node of grid cell g_i
$g_i.l_v$	The taxi list of grid cell g_i
$g_i.l_g^t$	The temporally-ordered grid list of grid cell g_i
$g_i.l_g^d$	The spatially-ordered grid list of grid cell g_i

B. The Dynamic Taxi Ridesharing Problem

In this study, we consider the dynamic taxi ridesharing problem defined as follows: *given a fixed number of taxis traveling on a road network and a stream of queries (i.e. a sequence of queries in ascending order of their birth time), we aim to serve each query Q in the stream by dispatching the taxi which satisfies Q with minimum additional incurred travel distance on the road network.*

The salient character of our problem definition lies in that we aim to minimize the increased travel distance for each individual query Q. This is obviously a greedy strategy and it does not guarantee that the total travel distance of all taxis for all queries is minimized. However, we still opt for this definition due to two major reasons.

First, the dynamic taxi ridesharing problem inherently resembles a greedy problem. In reality, taxi users usually expect that their requests can be served immediately. Given the rigid real-time context, the ridesharing service only has information of currently available queries and thus can hardly make optimized schedules based on a global scope, i.e. over a long time span.

Second, the problem of minimizing the total travel distance of all taxis for the whole query stream is NP-complete. We prove this statement as follows. The problem of optimizing travel distance for all taxis for the whole query stream, denoted by *Total Distance Optimization Taxi Ridesharing Problem (TDOTRP)*, can be formalized as the following decision problem: given a stream of queries S_Q, a start time t_s (t_s is the smallest value among the birth time of any query in S_Q) and a set of taxi statuses S_V at t_s, a road network RN in which each road segment is associated with a speed limit, a number $P \in [0,100]$ and a number $D \geq 0$, plan a schedule for each taxi such that the total travel distance of all taxis is no larger than D and the fraction of satisfied queries is at least P

precent. The TDOTRP is NP-complete because we can prove that it is a generalization of the *Travelling Salesman Problem with Time Window (TSPTW)*, which has already been proved to be NP-complete [19]. The input of a TSPTW instance includes a start time t_0, N vertices $\{1, 2,.., n\}$ in which vertex 1 is the depot vertex, the pair-wise distances between vertices and a number $D' \geq 0$. Each vertex i is also associated with a time window $i.w =< e_i, l_i >$, where $l_i \geq e_i \geq t_0$ for all $i = 1, \dots, n$. The question is to find out whether or not there is a cycle route of distance no larger than D' such that a salesman can leave the depot, i.e. vertex 1 at t_0, visit each vertex i ($i = 1, 2, \dots, n$) once within their corresponding time window and return to the depot.

An instance of the TDOTRP I_{TDOTRP} can be constructed from an instance of the TSPTW problem I_{TSPTW} by: (i) create the road network of I_{TDOTRP} using the vertex pair-wise distance of I_{TSPTW}; (ii) place one vacant taxi at vertex 1 and let the start time $t_s = t_0$; (iii) create a query Q_i for each vertex i such that $Q_i.o = Q_i.d = i$, and $Q_i.wp = Q_i.wd = i.w$, $Q_i.t = t_0$ for $i = 1, \dots, n$; In other words, every vertex i ($i = 1, \dots, n$) of I_{TSPTW} is considered as a dummy query of which the pickup point (time window) coincides the delivery point (time window) and the query is known a priori; (iv) let P=100, which means I_{TDOTRP} needs to satisfy all the queries, and $D = D'$.

The above construction completes the proof that TDOTRP is a generalization of TSPTW. Since TDOTRP is clearly in NP, therefore, we have proved that TDOTRP is NP-complete.□

C. Framework of the Dynamic Taxi Ridesharing Service

The framework of our dynamic taxi ridesharing service is shown in Fig. 1. As depicted by the broken red arrows, a taxi uploads its status V (defined in Definition 3) to the operation centre when joining in the ridesharing service, or when a passenger gets on or off the taxi, or at a frequency (e.g., every 20 seconds) while connected to the service. The system maintains spatio-temporal index of the taxis for the purpose of fast user query processing. The index will be updated once a new status of a taxi is received or a taxi's route is re-scheduled by the service.

A passenger submits a query Q (refer to Definition 1 for details) to the system and receives a response R_u from the service. As demonstrated by the solid blue arrows, all incoming queries of the system are streamed into a queue S_Q and are processed according to the first-come-first-serve principle. For each Q at the top of the query queue, the system invokes the *Taxi Searching* module to search for a set of candidate taxis $\{Taxis\}$ which is likely to satisfy the query based on the latest index. Given the result set $\{Taxis\}$, the system invokes the *Scheduling* module to insert the query into the schedule of a taxi in the set $\{Taxis\}$ which satisfies Q with minimum increase in travel distance. If the query is satisfied (see Definition 4), the service (i) informs the passenger with response R_u (which is comprised of the ID of the taxi scheduled to pick her up and the estimated pickup time); (ii) sends R_v (the new schedule) to the taxi, and updates the spatio-temporal index accordingly. Otherwise, the response R_u asks the passenger to modify the query or resubmit it later. An early response R_u may be sent to the passenger (as illustrated by the blue dotted arrow) if the taxi searching module returns an empty taxi set.

In addition, a pricing scheme is also designed to charge a ridesharing passenger properly, providing taxi drivers with more profit, and reducing the expense of each individual passenger compared to a single-passenger ride.

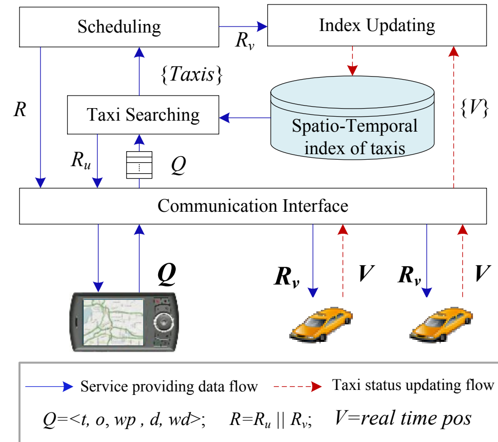


Fig. 1 Framework of the dynamic taxi ridesharing service

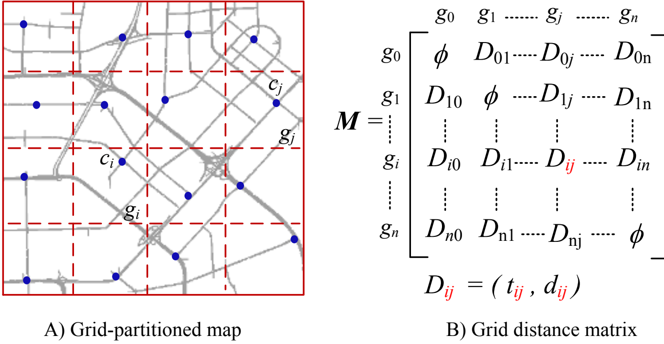
III. INDEX

Remember that the taxi searching module aims to quickly select a small set of taxis which is likely to satisfy the new query with a small increase in travel distance. It is easy to see that a necessary condition for a taxi to satisfy a query Q is that the taxi needs to be able to pick up the passenger of Q on time, i.e. before timestamp $Q.pw.l$. This observation naturally suggests us to look for taxis "near" the pickup point of Q. From this point of view, this problem resembles the *K Nearest Neighbour (KNN)* problem for moving objects [4] as both problems are interested in finding moving objects in the proximity of a static point. However, unlike in the KNN problem where the number of returned objects is explicitly determined by the given number K, the number of taxis to be retrieved here is not bounded by any fixed value but implicitly decided by the temporal constraints of the query. In other words, in this problem we need to use the query's time windows to filter out unsatisfactory taxis.

To achieve this goal, a straightforward approach as in [3][15] is that, for each taxi we calculate the shortest path between the current position of the taxi and the pickup point of the new query and see if the corresponding travel time is smaller than the available time, i.e. the difference between current time and $Q.pw.l$. Unfortunately, given the real-time context, this approach is too time-consuming because the shortest path computation is expensive and the number of taxis is huge. In other words, we want a taxi searching process that is both fast and selects taxis wisely such that the selected taxis can satisfy the query with a reasonable small increase in travel distance over their existing schedules. The problem of the straightforward approach lies in that it needs to calculate a

shortest path for each taxi, which is prohibitively expensive in time. What if we use a pre-computed distance to approximate the distance of the shortest path? Though the distance is not exact anymore, the time-consuming problem completely goes away. Inspired by this idea, we propose a spatio-temporal index of taxis. In the rest of this section, we describe how the index is built and updated.

We partition the road network using a grid. (Other spatial indices such as R tree can be applied as well, but we envision that the high dynamics of taxis will cause prohibitive cost for maintaining such an index.) As shown in Fig. 2 A), within each grid cell, we choose the road network node which is closest to the geographical centre of the cell as the *anchor* node of the cell (represented by a blue dot in Fig. 2 A). The anchor node of a grid cell g_i is thereafter denoted by c_i. We pre-compute the distance, denoted by d_{ij} , and travel time, denoted by t_{ij}, of the shortest path on the road network for each anchor node pair c_i and c_j. Quite a few advanced travel time prediction techniques [25] (e.g., incorporating real time traffic conditions) can be applied to estimate the travel time. However, since the traffic prediction is not a focus of this paper, we just use the speed limit of road segments to calculate travel time t_{ij} for the sake of simplicity. The distance and travel time results are saved in a matrix as shown in Fig. 2 B). The matrix is thereafter referred to as the *grid distance matrix*.



A) Grid-partitioned map B) Grid distance matrix

Fig. 2 Grid partitioned map and the grid distance matrix

Now imagine that each grid cell collapses to its anchor node, that is, all the points in one cell fall to its anchor node. Then the distance between any two arbitrary points equals to the distance between two corresponding anchor nodes. In other words, the grid distance matrix provides an approximated distance of the road network shortest path between any two geographical points at the grid cell level. Using this approximated distance, we can completely avoid the expensive shortest path calculation at the stage of taxi searching.

Each cell has some internal data structure for the purpose of taxi searching. Specifically, each grid cell g_i maintains three lists: a *temporally-ordered grid cell list* ($g_i.l_g^t$), a *spatially-ordered grid cell list* ($g_i.l_g^d$), and a *taxi list* ($g_i.l_v$), as illustrated in Fig. 3 A) (here we only describe how to build the lists and leave their usage in Sec. IV when we introduce the taxi searching algorithms).

$g_i.l_g^t$ is a list of other grid cells sorted in ascending order of the travel time from these grid cells to g_i (temporal closeness). Likewise, $g_i.l_g^d$ is a list of other grid cells sorted in ascending order of the travel distance to g_i (spatial closeness). The spatial and temporal closeness between each pair of grid cells are measured by the values saved in the grid distance matrix shown in Fig. 2 B). For example, t_{2i} measures the temporal closeness from g_2 to g_i , and d_{2i} measures the spatial closeness from g_2 to g_i.

These two grid cell lists are static. That is to say, they are only computed once. It is worth mentioning that cells that are neighbours in the grid may not be neighbours in a grid cell list because the distance is measured in the road network instead of a free space.

The taxi list $g_i.l_v$ of grid cell g_i records the IDs of all taxis which are scheduled to enter g_i in near future (typically within a temporal scope of one or two hours). Each taxi ID is also tagged with a timestamp t_a indicating when the taxi will enter the grid cell. All taxis in the taxi list are sorted in ascending order of the associated timestamp t_a . $g_i.l_v$ is updated dynamically. Specifically, taxi V_j is removed from the list when V_j leaves g_i; taxi V_k is inserted into the list when V_k is newly scheduled to enter g_i. If taxis are tracked (see [20]), when new GPS records are received from taxis, taxi lists need to be updated. Specifically, when a new GPS record from V_p is received, denote by g_q the current cell in which V_p is located, the timestamp associated with V_p in the taxi list of cell g_q and cells to be passed by V_p after g_q need to be updated.

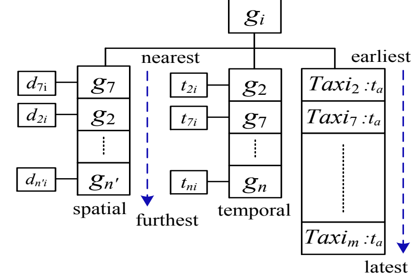


Fig. 3 Spatio-temporal index of taxis

IV. Taxi Searching Algorithms

A. Single-side Taxi Searching

Now we are ready to describe our first taxi searching algorithm. For the sake of the clarity of description, please consider the example shown in Fig. 4. Suppose there is a query Q and the current time is t_{cur}. g_7 is the grid cell in which $Q.o$ is located. g_7*'s* temporally-ordered grid cell list $g_7.l_g^t$ is shown on the right of Fig. 4. g_7 is the first grid cell selected by the algorithm. Any other arbitrary grid cell g_i is selected by the searching algorithm if and only if Eq. (1) holds, where t_{i7} represents the travel time from grid cell g_i to grid cell g_7. Eq. (1) indicates that any taxi currently within grid cell g_i can enter g_7 before the late bound of the pickup window using the travel time between the two grid cells (if we assume that each grid cell collapses to its anchor node).

$$t_{i7} + t_{cur} \leq Q.wp.l \quad (1)$$

To quickly find all grid cells that hold Eq. (1), the single-side searching algorithm simply tests all grid cells in the order-preserved list $g_7.l_g^t$ and finds the first grid cell g_f which fails to hold Eq. (1). Then all taxis in grid cells before g_f in list $g_7.l_g^t$ are selected as candidate taxis.

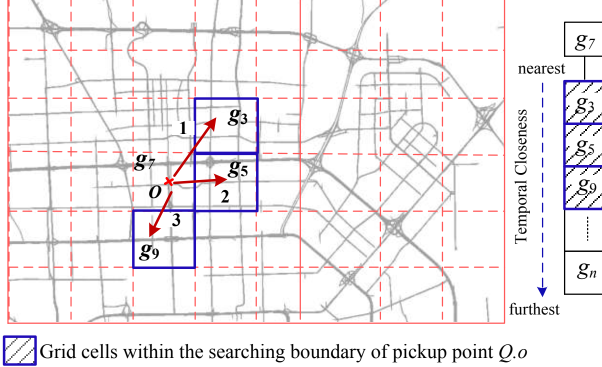


Fig. 4 The single-side taxi searching algorithm

In Fig. 4, grid cell g_3, g_5 and g_9 are selected by the searching algorithm. Then for each selected grid cell g_s, the algorithm selects taxis (in $g_s.l_v$) whose t_a is no later than $Q.wp.l - t_{s7}$. For instance, Fig. 5 shows how taxis are selected from grid cell g_7 and g_3.

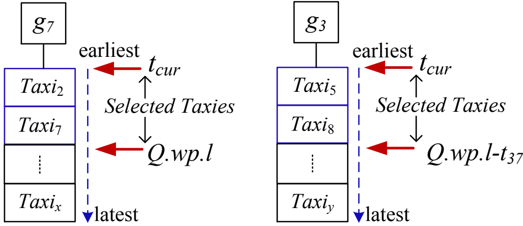


Fig. 5 Choose taxis from the selected grid cells

The taxi which can satisfy Q with the smallest increase in travel distance must be included in one of the selected grid cells (under the assumption that each grid cell collapses). Unfortunately, this algorithm only considers taxis currently "near" the pickup point of a query (thus called single-side search). As the number of selected grid cells could be large, this algorithm may result in many taxis retrieved for the later scheduling module (therefore increasing the entire computation cost), which is certainly not desirable for a rigid real-time application like taxi ridesharing. Actually, the spatiotemporal factor on the delivery point of queries also provides us with opportunities to reduce the number of grid cells to be selected. Along this idea, we propose a dual-side searching algorithm as an effort for striking a balance between the distance optimality and the computation cost.

B. Dual-Side Taxi Searching

At its core, the dual-side searching is a bi-directional searching process which selects grid cells and taxis from the origin side and the destination side of a query simultaneously.

To dive into the details of the algorithm, consider the query illustrated in Fig. 6 where g_7 and g_2 are the grid cells in which $Q.o$ and $Q.d$ are located respectively. Squares filled with stripes indicate the possible grid cells searched by the dual-side searching algorithm at $Q.o$ side. They are determined by the temporal closeness between a query and taxis (Refer to Eq. 1 for details). The red number in each such grid cell indicates its relative position in the spatially-ordered grid list of g_7. Squares filled with dots indicate the grid cells accessed by the dual-side searching algorithm at $Q.d$ side. Any grid cell g_j other than g_2 is selected by the searching algorithm at $Q.d$ side if and only if Eq. (2) holds, which means that any taxi currently in g_j can enter g_2 before the late bound of the delivery window (assumes that grid cells collapse).

$$t_{cur} + t_{j2} \leq Q.wd.l \quad (2)$$

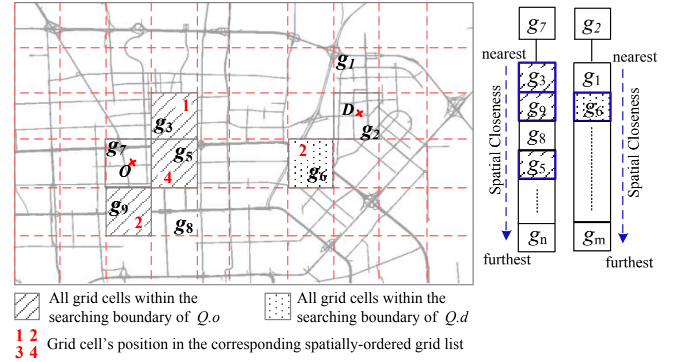


Fig. 6 Overview of the dual-side taxi searching algorithm

Similar to finding all grid cells in which Eq. (1) holds, all grid cells in which Eq. (2) holds can be quickly determined by scanning the temporally-ordered grid list $g_2.l_g^t$ in order. All grid cells within the searching boundary at $Q.d$ side are then considered. Fig. 6 shows g_6 is the only satisfying cell in this example.

Fig. 7 further illustrates the dual-side searching algorithm step by step. The algorithm maintains a set S_o and a set S_d to store the taxis selected from $Q.o$ and $Q.d$ side respectively. Initially, both S_o and S_d are empty. The first step in the searching is to add the taxis selected from taxi list $g_7.l_v$ to set S_o as depicted in Fig. 7 A), and add the taxis selected from taxi list $g_2.l_v$ to set S_d, as depicted in Fig. 7 B). Then the algorithm calculates the intersection of S_o and S_d. If the intersection is not empty, the algorithm stops immediately and returns the intersection set. Otherwise, it expands the searching area by including one other grid cell at each side at a time.

To select next cells, we use the following heuristic: for a taxi V, the closer some cell to be passed by V is to g_7 and the closer some cell to be passed by V is to g_2 (measured in the distance between the anchor nodes of the cells), the smaller the V's scheduled travel distance increases after insertion of the query. Thus, for the purpose of minimizing the increased travel distance, the next grid cell included at $Q.o$ side is chosen as the next element in the spatially-ordered grid list $g_7.l_g^d$ which holds Eq. (1). Similarly, the next grid cell

included at $Q.d$ side is always chosen as the next element in the spatially-ordered grid list $g_2.l_g^d$ which holds Eq. (2).

In this example, since the intersection of S_o and S_d produces an empty set, the algorithm expands at $Q.o$ side to include g_3 (indicated by the broken red rectangle) and adds taxis selected from $g_3.l_v$ as depicted in Fig. 7 C). At $Q.d$ side, the algorithm covers g_6 and adds taxis as indicated in Fig. 7 D).

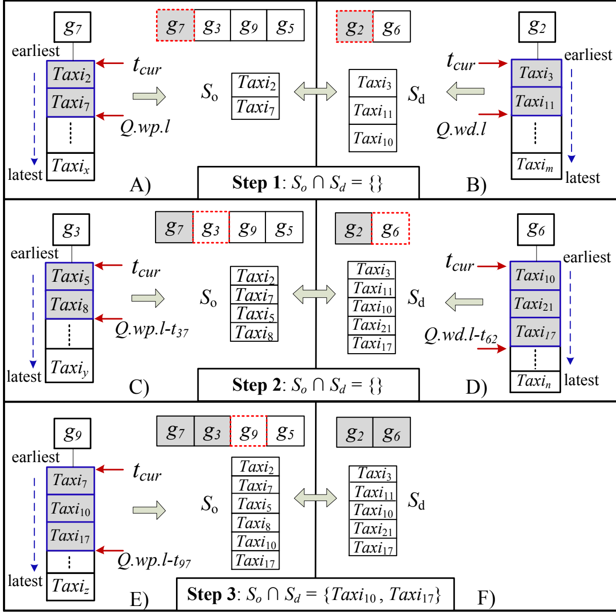


Fig. 7 Calculation of the taxi set in the dual-side searching

Unfortunately, the intersection set of S_o and S_d remains empty. Consequently, the algorithm continues to expand the searching area at both sides. Thus, g_9 is selected at $Q.o$ side; but no grid cell can be further included at the $Q.d$ side. After adding the taxis selected from $g_9.l_v$ into set S_o as shown in Fig. 7 E), we finally find $Taxi_{10}$ and $Taxi_{17}$ as the intersection between S_o and S_d. So the searching algorithm terminates.

The pseudo code of the dual-side searching is presented in Algorithm 1. The dual-side searching algorithm may not always find the taxi with the minimum travel distance increase for a query. However, as a compensation for the small loss in distance optimality, the algorithm selects far fewer taxis for the following scheduling module, therefore reducing the computation cost. We found in the experiments that the number of selected taxis is reduced by 50% while the increase in travel distance is just 1% over the single-side search algorithm.

V. Scheduling Module

Given the set of taxi statuses $\{Taxis\}$ retrieved for a query Q by the taxi searching algorithm, the purpose of the scheduling module is to insert Q into the schedule of the taxi which satisfies Q with minimum additional travel distance. In the rest of this section, Part A describes how to insert a query Q into the schedule of a taxi status V, and Part B introduces the lazy shortest path calculation strategy, which is used to speed up the computation involved in the scheduling.

Algorithm 1: Dual side taxi searching

Data: Query Q, the current time t_{cur}
Result: a set of taxies S

1 $g_o \longleftarrow$ the grid cell in which $Q.o$ locates
2 $S_o \longleftarrow$ taxies will be in g_o before $Q.wp.l$
3 $g_d \longleftarrow$ the grid cell in which $Q.d$ locates
4 $S_d \longleftarrow$ taxies will be in g_d before $Q.wd.l$
5 $S \longleftarrow S_o \cap S_d$
6 $l_o \longleftarrow \emptyset$
7 **for** *Grid cell* g_i *in* $g_o.l_g^t$ **do** /* set the searching boudary at $Q.o$ side */
8   **if** $t_{cur} + t_{io} \leq Q.wp.l$ **then** $l_o \longleftarrow l_o \cup \{g_i\}$
9   **else** break
10 $l_d \longleftarrow \emptyset$
11 **for** *Grid cell* g_j *in* $g_d.l_g^t$ **do** /* set the searching boudary at $Q.d$ side */
12   **if** $t_{cur} + t_{jd} \leq Q.wd.l$ **then** $l_d \longleftarrow l_d \cup \{g_j\}$
13   **else** break
14 $stop_o \longleftarrow False$ /* stop flag for origin side searching */
15 $stop_d \longleftarrow False$ /* stop flag for destination side searching */
16 **while** S *is* $\emptyset$ *and* ($stop_o = False$ *or* $stop_d = False$) **do**
17   **if** *not all grid cells in* l_o *have been visited* **then**
18         $g_i \longleftarrow$ the next grid cell in grid list $g_o.l_g^d$ which is also in l_o and has not been visited
19         $S_o \longleftarrow S_o \cup \{$ taxies will enter g_i before $Q.wp.l - t_{cur}\}$
20   **else** $stop_o$ =True
21   **if** *not all grid cells in* l_d *have been visited* **then**
22         $g_j \longleftarrow$ the next grid cell in grid list $g_d.l_g^d$ which is also in l_d and has not been visited
23         $S_d \longleftarrow S_d \cup \{$ taxies will enter g_j before $Q.wd.l - t_{cur}\}$
24   **else** $stop_d$ =True
25     $S \longleftarrow S_o \cap S_d$
26 return S

A. Insertion Feasibility Check

Given a new query Q and a taxi status V, consider how to determine whether or not Q can be inserted into $V.s$ and how to perform the insertion if appropriate. As in [8][16], here we assume that the order of points in the current schedule remains intact when inserting a new query to the schedule. Then at a high logical level, the insertion can be separated into two stages: (i) insert the pickup point of the query $Q.o$; (ii) insert the delivery point of the query $Q.d$. For example, Fig. 8 shows one possible way to insert query Q into schedule $Q_1.o \rightarrow Q_2.o \rightarrow Q_1.d \rightarrow Q_2.d$. Among all possible ways of insertion, the system chooses the insertion way that minimally increases the travel distance. So the scheduling module is thus able to choose the taxi V out of the taxi status set $\{Taxis\}$ which incurs the minimum increase in travel distance.

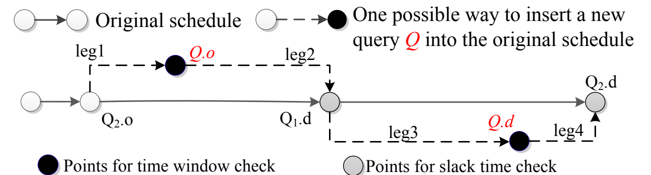


Fig. 8 One possible insertion of a query into a schedule

For each insertion possibility (uniquely identified by the position i and j at which $Q.o$ and $Q.d$ is inserted into the schedule), the system uses Algorithm 2 to evaluate its feasibility. We refer to each such invocation of Algorithm 2 as an *insertion feasibility check*. For instance, consider the example shown in Fig. 8. To insert $Q.o$ immediately after point $Q_2.o$, the algorithm first checks whether the corresponding taxi is able to arrive at $Q.o$ before $Q.wp.l$. If not, then the insertion fails. Otherwise, the algorithm then

computes the travel time delay t_d due to the insertion of $Q.o$ using Eq. (3), where $\rightarrow$ denotes the estimated dynamic travel time (e.g. using the technique proposed in [25]) from one location to another location, and t_w represents the time spent waiting for the passenger if the taxi arrives $Q.o$ early, i.e. ahead of $Q.wp.e$.

$$t_d = (Q_2.o \rightarrow Q.o) + (Q.o \rightarrow Q_1.d) + t_w - (Q_2.o \rightarrow Q_1.d) \quad (3)$$

If the time delay results the late arrival at any point after $Q.o$ in the original schedule, then the insertion fails. For this purpose, we introduce the notion of *slack time*. Denote by a_p and a_d the projected arrival time at the pickup point $Q.o$ and the delivery point $Q.d$. Then the slack time at $Q.o$ and $Q.d$, denoted by $(Q.o)._{st}$ and $(Q.d)_{st}$ respectively, is calculated by Eq. (4) and Eq. (5).

$$(Q.o)._{st} = Q.wp.l - a_p \quad (4)$$

$$(Q.d)_{st} = Q.wd.l - a_d \quad (5)$$

Thus, we can use the slack time as the shortcut to check whether the delay due to an insertion destroys the timely arrivals at any subsequent point in the schedule. As depicted in Fig. 8, points with grey background should be examined for the slack time check after the insertion of $Q.o$. That is, if $a_d \geq Min\{(Q_1.d)._{st}, (Q_2.d)._{st}\}$, then the insertion fails. If $Q.o$ is inserted successfully, the system proceeds to insert $Q.d$ in a similar way. The increased travel distance is calculated when both $Q.o$ and $Q.d$ are inserted successfully.

Algorithm 2: Insertion feasibility check

Data: Query Q, taxi status V, insertion position i for $Q.o$, insertion position j for $Q.d$, current time t_{cur}.
Result: Return *new_schedule* if the insertion succeeds; otherwise return False.

```
/* → represents travel time between two locations here */
1  if t_cur + (V.l → Q.o) > Q.wp.l then /* cannot arrive Q.o on time */
2      return False
3  if the time delay incurred by the insertion of Q.o causes the slack time of
   any point after position i in schedule V.s smaller than 0 then
4      return False
5  new_schedule ⟵ insert Q.o into V.s at position i /* the slack time
   of each pickup(delivery) point after position i is updated
   accordingly at the meantime */
6  t_j ⟵ the scheduled arrival time of the j^th point of V.s
7  l_j ⟵ the geographical location of the j^th point of V.s
8  if t_j + (l_j → Q.d) > Q.wd.l then /* cannot arrive Q.d on time */
9      return False
10 if the time delay incurred by the insertion of Q.d causes the slack time of
   any point after position j in new_schedule smaller than 0 then
11     return False
12 new_schedule ⟵ insert Q.d into new_schedule at position j /* the
   slack time of each pickup(delivery) point in new_schedule is
   updated accordingly at the meantime */
13 return new_schedule
```

Now let us consider the computation cost of the scheduling module. For each schedule $V.s$ composed of n points, there are at $n+1$ positions to insert the pickup point $Q.o$ and $n-i+1$ positions to insert the delivery point $Q.d$ into $V.s$, given that i is the position at which $Q.o$ is inserted. That is to say, there are $O(n^2)$ possible ways of insertion in total. And for each possible way of insertion, shortest path calculation is invoked as many as four times, as depicted in Fig. 8 where each labelled leg represents one shortest path calculation. Unfortunately, the cost of shortest path calculation is expensive in a real-time application like the dynamic taxi ridesharing studied here. Thus, accelerating the execution of the insertion feasibility check is critical to the scalability of the system.

Some existing work considers ignoring some insertion possibilities. For instance, [16] suggests two alternative insertion strategies. One strategy is only checking the insertion ways in which the query is inserted at the beginning of the schedule. That is, the system always requires the taxi to reroute for the pickup point of the new query right away. If the insertion fails, the taxi is no longer considered. Another strategy is to apply the optimization function to both stages of the insertion. For example, the pickup point is inserted as so to increase the travel distance minimally. When the delivery point is inserted, no other position for the pickup point will be considered. Though these strategies speed up the insertion process, it is not clear how much the quality of the chosen insertion will deteriorate. Thus we aim to expedite the insertion process by speeding up the calculation itself instead of eliminating some insertion possibilities.

B. Lazy Shortest Path Calculation Strategy

In this part, we propose a lazy shortest path calculation strategy that leverages the pre-computed grid distance matrix, the triangle inequality and caching to speed up the feasibility check process. The essence of the proposed strategy is to delay the shortest path calculation until the calculation is needed.

Recall that the road network is partitioned into grid cells. Whenever the insertion feasibility check is invoked, the calculation of shortest path is deferred or avoided by the following logic: if the shortest path between the origin location O and the destination location D has been previously calculated, then the algorithm simply retrieves the path from the cached results; otherwise, instead of directly calculating the shortest path, the algorithm first calculates the lower bound of the travel time between O and D using the pre-computed travel time between grid cells and the triangle inequality. For example, consider a taxi currently at point O and a new query with pickup point at D. Denote by g_i and g_j the grid cell in which O and D are located respectively. By applying the triangle inequality, we have Eq. (6), where $\rightarrow$ denotes the estimated dynamic travel time from one location to another location.

$$t_{OD} \geq t_{ij} - (c_i \rightarrow O) - (D \rightarrow c_j) \quad (6)$$

Since t_{ij} is pre-computed, and $(c_i \rightarrow O)$ and $(D \rightarrow c_j)$ are usually easy to calculate as the origin and destination pair is confined within one grid cell, the lower bound of d_{OD} can be obtained instantaneously. Given the lower bound, the feasibility of an insertion may be determined much quicker. If the time window constraint cannot be satisfied even for the lower bound of the shortest path, then the insertion way must be infeasible. Only when the lower bound does not violate time constraints, the algorithm needs to proceed to compute

the shortest-time path between points O and D. Clearly, the lower bound delays the shortest path calculation until the time when the calculation is absolutely needed. In addition, state-of-art shortest path algorithms, e.g. [10], can be applied to speed up on-line shortest path calculations.

It is evident that the grid size will affect the effectiveness of lazy shortest path calculation strategy. On one hand, if the granularity of the grid is too coarse, i.e. each grid cell is too large, the routing within a grid cell will be expensive, which defeats the primary purpose of the lazy strategy; on the other hand, if the granularity of grid is too fine, i.e. each grid cell is too small, the cost of updating the taxi list of the grid cells, i.e. updating the timestamps when taxis are going to enter the cells, will be high.

C. Pricing Scheme

As our goal in this study is to propose a complete practical taxi ridesharing service, we provide a simple yet effective pricing scheme. We believe that it is reasonable to assume following properties for a pricing scheme: (i) taxi fare per mile is higher for multiple passengers than for a single passenger; (ii) the taxi fare of shared distances is evenly split among the riding passengers; as a result, the more people share a ride, the lesser each individual pays for the ride.

Based on these two properties, we propose the following pricing scheme. Denote by p the regular taxi fare per mile. Let $\alpha \in (0,1)$ be the fare inflating parameter, that is, the taxi fare per shared mile is $(1+\alpha)p$. The taxi fare for shared miles is evenly split among all passengers involved. Consequently, the taxi fare for each passenger is likely to be reduced if she shares a certain distance during the trip. The taxi fare of each passenger can be then automatically calculated by Eq. (7), where d_m is the travel distance shared by m passengers, and c is the capacity of the taxi.

$$Fare = p(d_1 + \sum_{m=2}^{c} (\alpha+1) * d_m/m) \tag{7}$$

On the other hand, the total fare for all taxi drivers is calculated by Eq. (8), where D_n is the total travelled distance that is not shared and D_r is the total travelled distance that is shared. We will examine the appropriate value for α to make ridesharing profitable for taxi drivers.

$$Total_Profit = p(D_n + (1+\alpha) * D_r) \tag{8}$$

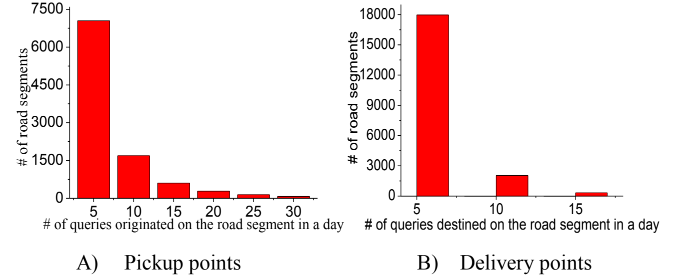
VI. Evaluation

A. Setting

1) Dataset

Road networks: We perform the experiments using the real road network of Beijing, which contains 106,579 road nodes and 141,380 road segments.

Taxi Trajectories: The taxi trajectory dataset contains the GPS trajectory recorded by over 33,000 taxis during a period of 87 days spanning from March to May in the year of 2011. Each of 87 days has a single file contains all the trajectories recorded during the day. The total distance of the dataset is more than 400 million kilometres and the number of points reaches 790 million. After trip segmentation, there are in total 20 million trips, among which 46% are occupied trips and 54% are non-occupied trips. We map each occupied trip to the road network of Beijing using the map-matching algorithm proposed in [24]. Each trip then can be viewed as a query with windows size equals to 0. Fig. 9 shows the distribution of pickup and delivery points of the queries in the dataset over road segments in a day. It is clear that queries are distributed sparsely over the road network.



A) Pickup points B) Delivery points

Fig. 9 Distribution of queries over road segments

2) Experimental Platform

The historical trajectory dataset conceals rich information regarding 1) the distribution of the queries on the road network over time of day, and 2) the mobility patterns of the taxis. In order to validate our proposed service under practical settings, we mine the trajectory dataset to build an experimental platform, which generates a realistic query stream and initial taxi statuses for our experiments. We envision that this platform can be applied to many other relevant urban and transportation computation problems.

Query Stream: The goal is to generate dynamic queries that are as realistic as possible. For this purpose, we first discretise one day into small time bins, denoted by b_i and denote all road segments by r_i. We assign all historical queries into time bins based on the birth time of queries. Assume that the arrivals of queries on each road segment approximately follow a Poisson distribution during time frame f_j, where each frame has a fixed length spanning N time bins. Thus, we can learn λ_{ij} , i.e. the parameter of the Poisson distribution for road segment r_i during time frame f_j. Specifically, for each road segment r_i, we count the number of queries that originated from r_i within time frame f_j, denoted by c_{ij}, and learn the distribution of the destination road segment of these queries, denoted by p_{ij}. Then we calculate λ_{ij} based on c_{ij} using Eq. (*9*) and generate a query stream that follows a Poisson process with parameter λ_{ij}.

$$\lambda_{ij} = c_{ij} / N \tag{9}$$

For each query Q generated in frame f_j with the origin road segment being r_i, the destination road segment is generated according to the distribution p_{ij}. $Q.pw.e$ and $Q.dw.e$ equals to $Q.t$, i.e. the birth time of the query. $Q.pw.l$ is calculated by applying a fixed window size. $Q.dw.l$ equals to the sum of the late bound of the pickup time window and the average travel time between the origin and destination pair learned from the GPS trajectory dataset.

Note that the taxi GPS trajectory dataset only reveals the number of queries that got served. In reality there are also many queries unsatisfied and disappeared due to the shortage of taxis. In order to take such queries into consideration, we introduce a system parameter Δ, which stands for the ratio between the actual total number of queries and the number of queries extracted from the historical data. The inflated number of queries therefore equals to the number of queries extracted from the trajectory dataset multiplying Δ. We refer Δ to as the *query inflation multiplier* thereafter. Fig. 10 shows how the inflated number and the extracted number of queries fluctuate during a day (the time frame is 1 hour and Δ=2 in the figure).

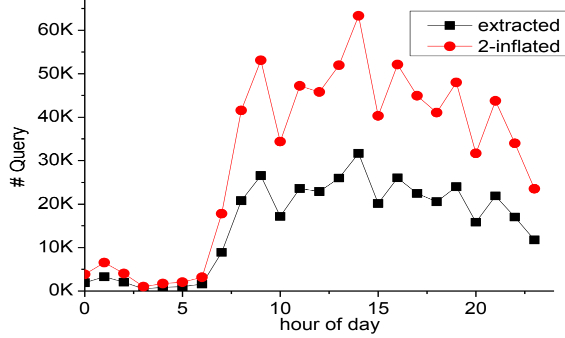


Fig. 10 Inflated and extracted number of queries during a day

Initial Taxi Statuses: To keep the characteristics of the realistic scenario, we use the real taxi statuses by slicing the historical trajectories at a certain timestamp. Specifically, we select a date and choose a particular second of day as the timestamp when the experiment starts, denote it by t_s. We scan all the GPS records of the selected date to determine the initial states of taxis. A taxi status V is set to be occupied if it is recorded occupied crossing timestamp t_s .The initial schedule of V can be initialized according to the record. A taxi V is set to be vacant if it is recorded vacant both just before and right after t_s. The concept of "just before" and "right after" is controlled by a temporal parameter, which is set to be 2 minute. All remaining taxis are then considered as not recorded and thus not used in the simulation.

According to the setting mentioned above, we built a prototype system whose screenshot is shown in Fig. 11. The red polyline stands for the route that has been traversed in a schedule, the green part for the route to be travelled, and a person pin aligned with a "+" and "-" symbol for the pickup and delivery point of a query, respectively.

Fig. 11 A screenshot of T-Share prototype showing the route of a taxi

3) *Framework*

The set of queries and initial states used in all validation experiments are generated with parameters listed in TABLE II.

We first conduct experiments comparing the performance of the non-ridesharing method and different dynamic ridesharing methods (described soon) by varying the value of Δ. We also evaluate and analyse the efficiency and scalability of the ridesharing service by comparing competing ridesharing methods. Then we verify the benefit of the lazy shortest path calculation strategy in reducing computation cost, given different grid sizes. Finally, we investigate how the profit of taxi drivers affected under the proposed pricing scheme.

TABLE II PARAMETER SETTING FOR QUERY GENERATION

Notation	Definition	Value
t_s	The start time of simulation	9 am
t_e	The end time of simulation	9:30 am
$\#taxi$	The number of taxis	2,980
ws	The pickup window size	5 minute
$l(b_i)$	The length of a time bin	5 minute
N	The # of time bins in a frame	12

4) *Measurements*

The performance of the ridesharing service is evaluated by following measures.

Relative Distance Rate (RDR): Define the distance of a query Q as the distance between its pickup point $Q.o$ and its delivery point $Q.d$. Denote by D_{SQ} the sum of distances of queries that get satisfied and by D_V the total distance travelled by all taxis in the ridesharing. *RDR* is calculated by Eq. (10).

$$RDR = D_V / D_{SQ} \quad (10)$$

RDR evaluates the effectiveness of ridesharing by measuring how much distance is saved compared to the case where no ridesharing is practiced.

Satisfaction Rate (SR): is the fraction of queries get satisfied in the ridesharing (exclude queries that are already served by taxis at the initial state in the query counting). *SR* is a crucial criterion measuring the effectiveness of the ridesharing system.

Number of Road Nodes Accessed Per Query (#RNAPQ): is the number of accessed road network nodes per query.

Number of Grid Cells Accessed Per Query (#GCAPQ): is the number of accessed grid cells per query.

Number of Taxis Accessed Per Query (#TAPQ): This measurement records how many taxis per query are accessed for insertion feasibility checks by the scheduling module. *RNAPQ*, *GCAPQ*, *TAPQ* are indicators for computation cost of the system since the majority on-line computation is done in the scheduling.

5) *Competing Methods*

We compare the performance of one non-ridesharing method and four flavours of our proposed ridesharing method.

The *Non-Ridesharing method (NR)* forbids ridesharing and assumes that when a taxi becomes vacant, it moves towards the passenger that it can pick up at the earliest time among all queries it can satisfy. Since the scheduling is done by the central server, there is no need to worry about the competition between taxis.

We create four flavours of ridesharing method based on two choices, i.e. one choice made in the taxi searching step and the other choice made in the scheduling step. Specifically, a ridesharing method is said to be *dual-side* if the dual-side algorithm is used in the taxi searching step; otherwise, it is said to be *single-side* if the single-side algorithm is used. A ridesharing method is said to be *best-fit* if the scheduling module tries all taxis in result set returned by the taxi searching algorithm; otherwise, it is said to be *first-fit* if the scheduling module terminates immediately once it finds a taxi satisfying the query. Because the two choices can be made independently, we get the following 4 ridesharing methods: S*ingle-side and First Fit Ridesharing (SF)*, *Single-side and Best-fit Ridesharing (SB), Dual-side and First Fit Ridesharing (DF)*, *Dual-side and Best-fit Ridesharing (DB).*

B. Experiment Results

1) Effectiveness of the Ridesharing Service: We compare the satisfaction rate and the relative distance rate for ridesharing methods by varying the value of query inflation multiplier Δ. Meanwhile the road network is divided into 30*30 cells.

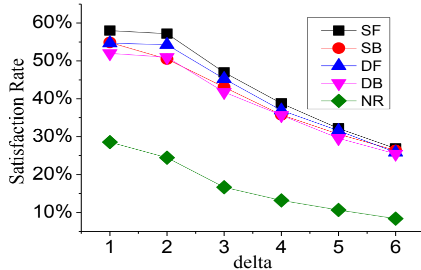


Fig. 12 Satisfaction rate vs. query inflation multiplier

As Fig. 12 shows, all ridesharing methods have a considerably higher satisfaction rate (about 25% higher on average) than the *NR* method for all delta values. The difference between ridesharing methods on the satisfaction rate is insignificant as no particular technique is proposed for optimizing the satisfaction rate.

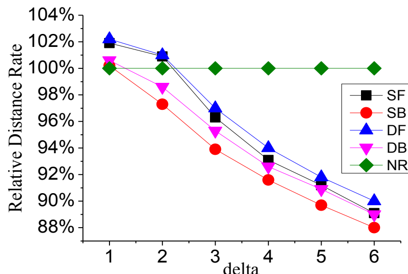


Fig. 13 Relative Distance Rate vs. query inflation multiplier

Fig. 13 shows how *RDR* changes over Δ for different ridesharing methods. *RDR* steadily drops as the delta value increases. This is likely because the taxi ridesharing opportunities surge as the number of queries increases. The *SB* ridesharing method outperforms other methods, since *SB* reduces the travel distance increment the most. The *DB* method slightly trails the *SB* method as its taxi searching step explores fewer grid cells and taxis. In comparison, the two first-fit methods show a higher relative distance rate, especially when delta is small.

From the picture, we can see that ridesharing methods save up to 13% in travel distance, depending on delta. Given the fact that there are 67,000 taxis in Beijing and each taxi runs 480 km per day (learned from the dataset), the saving achieved by ridesharing here means over 1.6 billion kilometres in distance per year, which equals to 120 million liter of gas per year (supposing a taxi consumes 8 liter gasoline per 100km) and 2.3 million of carbon dioxide emission per year (supposing each liter of gas consumption generates 2.3 kg carbon dioxide).

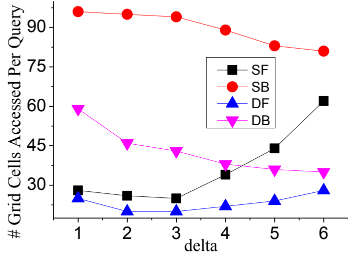
2) Scalability and Efficiency of the Ridesharing Service: We evaluate the scalability of the proposed ridesharing service by examining how the computation cost changes as the number of queries increases. The computation cost is measured by the average number of nodes (grid cells or road nodes) the system accesses for each query.

The three sub-graphs of Fig. 14 show the number of grid cells accessed per query, the number of road nodes accessed per query, and the number of taxis accessed per query for different ridesharing methods under various delta values. It is clear from the pictures that all ridesharing methods do not show sharp increase in computation cost as Δ increases.

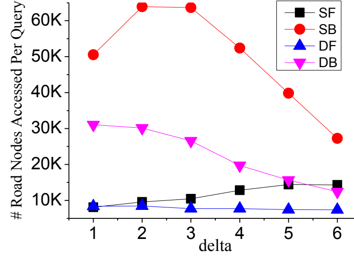
It is also obvious that the computation cost of the *DB* ridesharing method is significantly smaller than that of the *SB* method, actually even smaller than that of the *SF* method sometimes. The result of Fig. 13 and Fig. 14 together validate our motivation for the dual-side taxi searching algorithm. That is, the dual-side searching indeed incurs a small increase in travel distance, in exchange for the significant decrease in computation cost.

3) Effectiveness of the Lazy Shortest Path Calculation Strategy: In this experiment, we validate effectiveness of the lazy shortest path calculation strategy in reducing the computation cost of the system as the grid size varies for Δ = 2. Fig. 15 shows how the number of road nodes accessed per query is changed for the *DB* ridesharing method with and without applying the lazy strategy. In both cases, previous calculated shortest path results are cached to avoid unnecessary repeated computation. Not surprisingly, the computation cost of the *DB* method is decreased by 83% on average when the lazy strategy is applied.

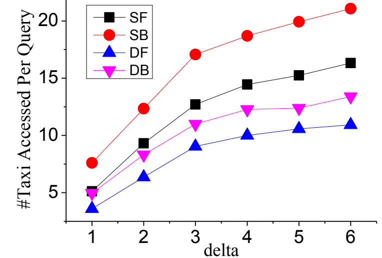
4) Pricing scheme: By applying Eq. (8), we compare the total profit of all taxi drivers with and without ridesharing. Fig. 16, A) and B) show the ratio of the profit with ridesharing compared to the profit without ridesharing, when the fare inflating ratio α equals 0.5 and 0.8 respectively. It is clear from the figures that all ridesharing methods make more profit than where no ridesharing is allowed. This result suggests that ridesharing can provide monetary incentives to drivers.



A) #GCAPQ vs. delta



B) #RNAPQ vs. delta



C) #TAPQ vs. delta

Fig. 14 Computation cost vs. query inflation multiplier

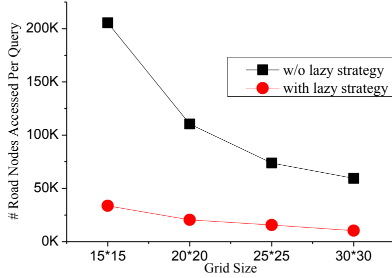
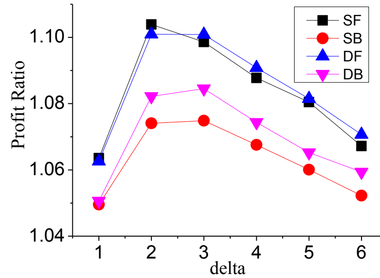
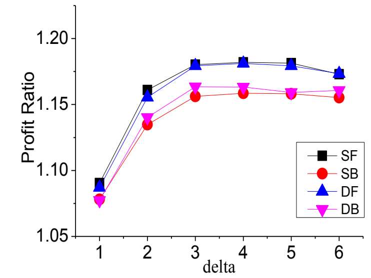


Fig. 15 Effectiveness of the lazy shortest path calculation strategy



A) α=0.5



B) α=0.8

Fig. 16 Profit ratio for ridesharing methods

VII. Related Work

We study three categories of related works, positioning our work in the research community.

A. Taxi Recommender and Dispatching Systems

Quite a few recommender systems have been proposed for improving an individual taxi driver's income and reducing unnecessary cruising. Based on historical taxi trajectories, Yuan and Zheng et al. [23][24] proposed a system that suggests some parking places for an individual taxi driver towards which they can find passengers quickly and maximize the profit of the next trip. Similarly, Ge et. al [14] suggests a sequence of pickup points for a taxi driver. While these systems are only designed from the perspective of taxi drivers, our service considers the needs of both taxi drivers and users.

Taxi dispatching services [22][18] usually send a taxi close to a passenger as per the passenger's call without considering ridesharing. Consequently, only vacant taxis need to be examined for each dispatch, which can be easily retrieved by answering a range query. In our case, each taxi that is occupied under full capacity needs to be considered. This complication introduces new challenges. Our work is also an example of urban computing [25], where a series of research work has been done with taxi trajectories.

B. Recurring Ridesharing

Recurring ridesharing deals with routine commutes. There are already existing websites and mobile applications for this purpose, such as Avego. Given the usual small size of the problem, researchers are able to solve it optimally by using linear programming techniques [6][2]. Compared to recurring ridesharing where queries are static, i.e., routes and time schedules are known in advance, the dynamic taxi ridesharing problem we studied here is more challenging, as queries are generated in real time and the routes of taxis change continuously.

C. Dial-A-Ride Problem

The taxi ridesharing problem can be viewed as a special member of the general class of the Dial-a-Ride Problem (DARP) [1], a.k.a. Vehicle Routing Problem with Time Windows [11]. DARP is essentially a constraint satisfaction problem, i.e., planning schedules for vehicles, subject to the time constraints on pickup and delivery events. The DARP is originated from and has been studied in various transport scenarios, notably goods transport [13], paratransit for handicapped and elderly personnel [3], etc. To the best of our knowledge, this work is the first to consider the DARP in the taxi ridesharing setting with dynamic queries.

Existing works on the DARP have primarily focused on the the static DARP, where all customer queries are known a priori. Since the general DARP is NP-hard [15], only small instances (involving only a few cars and dozens of queries) can be solved optimally (often by resorting to integer programming techniques, see [9][17]). Large static DARP instances are usually solved by using the two-phase scheduling strategy [5][8] with heuristics. Specifically, in phase I, queries are clustered and each cluster is assigned to a vehicle, i.e. forms an initial schedule. In phase II, queries are swapped among initial schedules of vehicles using heuristics.

Considerably less research has been carried out on the dynamic DARP, where customer queries are generated on the fly. Previous few works on the dynamic DARP problem [7][1][16] continue to adapt the two-phase scheduling strategy. As a result of dynamic queries, the two phase strategy now needs to be applied to each single query instead of the whole query pool as done in the static DARP. Particularly, phase I needs to select a taxi for a given query instead of clustering queries. In this sense, our approach resembles the phase I. Existing approaches to phase I in literature have been

discussed and compared with our approach at the beginning of Section IV. In addition, we believe that phase II is no longer suitable for a practical real-time taxi ridesharing service. The reason is that once a query is scheduled, the information of the dispatched taxi will be provided to the passenger. Swapping already scheduled, but not yet executed queries (i.e. passengers of the query not picked up yet) among taxis is likely to perplex passengers and severely damage the user experience of the ridesharing service.

CONCLUSIONS

This paper proposes a practical large-scale taxi ridesharing service. We evaluated our service based on a GPS trajectory dataset generated by 33,000 taxis over 3 months, in which over 10 million queries were extracted. Based on this data, we implemented a T-Share prototype system. The experimental results demonstrated the effectiveness and efficiency of our system in serving dynamic queries. Firstly, our service can enhance the delivery capability of taxis in a city so as to satisfy the commute of more people. For instance, when the ratio between the number of taxi queries and the number of taxis is 6, our service served additional 25% taxi users than no ridesharing. Secondly, compared with the taxi system sending passengers individually, our ridesharing service saves the total travel distance of taxis when delivering passengers, e.g., our approach saved 13% travel distance with the same ratio mentioned above. Supposing a taxi consumes 8 liters of gasoline per 100km, and given the fact learned from the real trajectory dataset that the average travel distance of a taxi in a day (in Beijing) is about 480km, our service can save over 120 million liter of gasoline per year (worth about 160 million dollar). Thirdly, our service can also save the expense of a taxi user, while increasing the profit of a taxi driver. Using the pricing scheme designed (when $\alpha = 0.8$), taxi drivers can increase their profit by 19% on average over no ridesharing. In addition, the experimental results justified the importance of the dual-side searching algorithm and the lazy shortest path calculation strategy. Compared to the single-side taxi searching algorithm, the dual-side taxi searching algorithm reduced the computation cost by over 50%, while the increase in travel distance was only about 1% on average. The computation cost was reduced by 83% if the lazy strategy is applied. On average, our service can answer a user query in 5ms even on a single machine, i.e., it can serve 720k queries per hour.

In the future, we will incorporate advanced travel time estimation techniques to improve the prediction of taxi travel time. We will also schedule queries that arrive within a small time interval in batch mode, to further optimize the total travel distance in ridesharing.

REFERENCES

[1] A Attanasio, J. F. Cordeau, G Ghiani, and G Laporte, “Parallel tabu search heuristics for the dynamic multi-vehicle dial-a-ride problem,” *Parallel Computing*, vol. 30, pp. 377-387, March 2004.

[2] R. Baldacci, V. Maniezzo, and A. Mingozzi, “An Exact Method for the Car Pooling Problem Based on Lagrangean Column Generation,” *Operation Research*, INFORMS, vol. 52, pp. 422-439, 2004.

[3] A. Beaudry, G. Laporte, T. Melo, and S. Nickel, “Dynamic transportation of patients in hospitals,” *OR Spectrum*, 2010.

[4] Rimantas Benetis, S. Jensen, Gytis Karciauskas, and Simonas Saltenis, “Nearest and reverse nearest neighbor queries for moving objects”, *The VLDB Journal* 15, 3, pp. 229-249, 2006.

[5] R. W. Calvo, and A. Colorni, “An effective and fast heuristic for the dial-a-ride problem,” *4OR: A Quarterly Journal of Operations Research*, vol. 5, pp. 61–73, 2007.

[6] R. W. Calvo, F. de Luigi, P. Haastrup, and V. Maniezzo, “A distributed geographic information system for the daily carpooling problem,” *Computer Operation Research*, Elsevier Science Ltd., 2004.

[7] A. Colorni, and G. Righini, “Modeling and optimizing dynamic dial-a-ride problems,” *International Transactions in Operational Research*, vol. 8, pp. 155–166, 2001.

[8] J. F. Cordeau, and Gilbert Laporte, “A tabu search heuristic for the static multi-vehicle dial-a-ride problem,” *Transportation Research Part B: Methodological*, vol. 37, pp. 579-594, July 2003.

[9] J. F. Cordeau, “A branch-and-cut algorithm for the dial-a-ride problem,” *Operation Research,* vol. 54, pp. 573–586, 2006.

[10] D. Delling, A. V. Goldberg, R. F. Werneck. Faster Batched Shortest Paths in Road Networks. *ATMOS, vol.20,* page 52-63, Germany, 2011.

[11] M. Desrochers, J. Lenstra, M. Savelsbergh, and F. Soumis, “Vehicle routing with time windows: optimization and approximation,” *Vehicle Routing: Methods and Studies*, Amsterdam, pp. 65–84, 1988.

[12] P. d’Orey, R. Fernandes, M. F. Empirical evaluation of a dynamic and distributed taxi-sharing system. In *IEEE Conf. on Intelligent Transportation Systems,* vol. 1, September, 2010.

[13] Y. Dumas, J Desrosiers, and F Soumis, “The pickup and delivery problem with time windows,” *European Journal of Operational Research*, Elsevier, vol. 54, pp.7-22, September, 1991.

[14] Y. Ge, H. Xiong, A. Tuzhilin, K. Xiao, M. Gruteser, and M. Pazzani, “An energy-efficient mobile recommender system,” In *KDD 2010*.

[15] P. Healy, and R Moll, “A new extension of local search applied to the Dial-A-Ride Problem,” *European Journal of Operational Research*, vol. 83, pp. 83-104, May 1995.

[16] M. Horn, “Fleet scheduling and dispatching for demand-responsive passenger services,” *Transportation Research Part C: Emerging Technologies*, vol. 10, pp. 35-63, February 2002.

[17] L. M. Hvattum, A Løkketangen, and G Laporte, “A branch-and-regret heuristic for stochastic and dynamic vehicle routing problems”, *Networks*, vol. 49, pp. 330–340, 2007.

[18] J. L. Lu, M.Y. Yeh, Y. C. Hsu, S. N. Yang, C. H. Gan, M. S. Chen, "Operating Electric Taxi Fleets: A New Dispatching Strategy with Charging Plans," In Proc. of *IEVC-2012*.

[19] M. W. P. Savelsbergh. Local search in routing problems with time Windows. *Annals of Operations Research*, vol. 4, pages 285–305, 1985.

[20] O. Wolfson, P. Sistla, B. Xu, J. Zhou, S. Chamberlain, Y. Yesha, N. Rishe, “Tracking moving objects using database technology in DOMINO”, *Proc. of NGITS*, 1999.

[21] Z. Xiang, C. Chu, and H. Chen, “A fast heuristic for solving a large-scale static dial-a-ride problem under complex constraints,” *European Journal of Operational Research*, vol. 174, pp. 1117–1139, 2006.

[22] K. Yamamoto, K. Uesugi, and T. Watanabe, “Adaptive routing of cruising taxis by mutual exchange of pathways,” In *Knowledge-Based Intelligent Information and Engineering Systems*, Springer, 2010.

[23] J. Yuan, Y. Zheng, L. Zhang, X. Xie, and G. Sun, “Where to find my next passenger,” In *Proc. of UbiComp 2011*, ACM, 109-118.

[24] J. Yuan, Y. Zheng, C. Zhang，X. Xie and G. Sun, “An Interactive-Voting based Map Matching Algorithm,” In Proc. of *MDM 2010*.

[25] J. Yuan, Y. Zheng, X. Xie, and G. Sun, “Driving with Knowledge from the Physical World,” In Proc. of *KDD 2011*.

[26] N. J. Yuan, Y. Zheng, L. Zhang, X. Xie, “T-Finder: A Recommender System for Finding Passengers and Vacant Taxis”. IEEE TKDE, 2013

[27] Y. Zheng and X. Zhou, *Computing with Spatial Trajectories*, Springer 2011.

[28] Y. Zheng, Y. Liu, J. Yuan, X. Xie, “Urban Computing with Taxicabs”, In *Proc. of UbiComp 2011*, ACM.