Top-k Taxi Recommendation in Realtime Social-Aware Ridesharing Services

Xiaoyi Fu[1(✉)], Jinbin Huang[1], Hua Lu[2], Jianliang Xu[1], and Yafei Li[3]

[1] Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong
{xiaoyifu,jbhuang,xujl}@comp.hkbu.edu.hk
[2] Department of Computer Science, Aalborg University, Aalborg, Denmark
luhua@cs.aau.dk
[3] School of Information Engineering, Zhengzhou University, Zhengzhou, China
ieyfli@zzu.edu.cn

Abstract. Ridesharing has been becoming increasingly popular in urban areas worldwide for its low cost and environment friendliness. In this paper, we introduce social-awareness into realtime ridesharing services. In particular, upon receiving a user's trip request, the service ranks feasible taxis in a way that integrates detour in time and passengers' cohesion in social distance. We propose a new system framework to support such a social-aware taxi-sharing service. It provides two methods for selecting candidate taxis for a given trip request. The grid-based method quickly goes through available taxis and returns a relatively larger candidate set, whereas the edge-based method takes more time to obtain a smaller candidate set. Furthermore, we design techniques to speed up taxi route scheduling for a given trip request. We propose travel-time based bounds to rule out unqualified cases quickly, as well as algorithms to find feasible cases efficiently. We evaluate our proposals using a real taxi dataset from New York City. Experimental results demonstrate the efficiency and scalability of the proposed taxi recommendation solution in real-time social-aware ridesharing services.

1 Introduction

Taxis play an important role as a transportation alternative between public and private transportations all over the world, delivering millions of passengers to their destinations everyday. Unfortunately, the number of taxis usually cannot meet the needs of people at peak time especially in urban areas, so that many people have to wait for a long time to grab a taxi. As a result, taxi ridesharing, which can reduce energy consumption and air pollutant emission, is considered as a promising solution to tackle this problem [1–4].

Many challenges exist in accomplishing a realtime taxi ridesharing service. From the riders' perspective, a rider seeks to share a ride with others with small detour and social comfort (e.g., having common friends or hobbies with other riders). Hence, if the service provider would like to encourage more users to

M. Gertz et al. (Eds.): SSTD 2017, LNCS 10411, pp. 221–241, 2017.
DOI: 10.1007/978-3-319-64367-0_12

participate in ride-sharing, it has to improve the service quality by striking a balance between riders' social connections and their travel distances. In real life, the safety concern of sharing a ride with strangers may hinder the user from accepting taxi ridesharing. Moreover, it may be uncomfortable and awkward for passengers to travel with someone they do not know at all, especially when the trip is long. It is therefore of high interest to provide taxi rideshare services that consider both social factor and trust issue as well as offer several good options for users to choose the taxi they would like to take.

The other challenge of a realtime ridesharing service is to process ride requests in real time. Recently, the popularization of GPS-enabled mobile devices has enabled people to call a taxi anywhere and at any time. This involves two tasks: (i) finding the taxis that are able to accommodate the new request without violating the constraints of their current schedules and (ii) selecting several good taxis (e.g., top-k taxis) for the user to choose. Given a large number of taxis and a new trip request, it is time consuming to find the feasible schedules and calculate riders' social connections at the same time. Therefore, designing an efficient and scalable ridesharing algorithm with a proper metric to measure whether a taxi is suitable for a given ride request is very challenging.

The majority of previous studies [3–5] focus on designing efficient assignment algorithms with the objective of minimizing the travel/detour cost and maximizing the rate of matched requests. Companies like *Uber*, *Didi* and *Kuaidi* allow riders to submit ride request ahead of time. However, these previous works have not fully met the new requirements, e.g., improving the ridesharing experience by considering the social aspect. Intrinsically, people would like to share a ride with someone who makes them feel comfortable and safe. This paper is concerned with *social-aware* ridesharing which aims to improve the ridesharing experience.

To address the aforementioned challenges, we propose a novel type of dynamic ridesharing service, named *top-k Social-aware Taxi Ridesharing (TkSaTR)* service, which processes user requests not only based on route schedule, but also considering riders' social relationships. Generally, given a set of taxis moving in a road network, each taxi has zero or several riders. Upon receiving a new trip request, the system provides the top-k qualified taxis for the rider to select by considering both the spatial and social aspects. If no appropriate taxi is retrieved, the system asks the user to modify the trip request or resubmit it at a later time.

The rest of this paper is organized as follows. Section 2 surveys the related work. The *TkSaTR* problem is formulated in Sect. 3. We present two candidate taxis searching methods in Sect. 4. In Sect. 5, we present the taxi scheduling and top-k taxi selection approach. Experimental results are reported in Sect. 6, followed by the conclusion and future work in Sect. 7.

2 Related Work

Existing studies on ridesharing fall into three categories: static ridesharing, dynamic ridesharing and trust-conscious ridesharing.

2.1 Static Ridesharing

Most relevant early works studied static ridesharing where the information of drivers and ride requests are known in advance. Static ridesharing covers three typical application scenarios: slugging, carpooling, and dial-a-ride.

In slugging [6], a rider walks to the driver's origin location and departs at the driver's departure time. At the driver's destination, the rider alights and walks to her/his own destination. Ma et al. [1] studied the slugging problem and its generalization from a computational perspective.

Carpooling is a typical ridesharing form for daily commutes where drivers need to adapt their routes to riders' routes. The carpooling problem is proved to be NP-hard, and the exact methods can only work efficiently on small instances, e.g., it can be solved by using linear programming techniques [7,8]c. For large instances of the carpooling problem, heuristics are proposed [9–11]. For a many-to-many carpooling system in which there are multiple vehicle and rider types, Yan and Chen [12] proposed a time-space network flow technique to develop a solution based on Lagrangian relaxation.

In the dial-a-ride problem (DARP), no private car is involved and the transportation is carried out by non-private vehicles (such as taxis) that provide a shared service. In order to receive the service, each rider submits a request with the origin and destination locations. In turn, the service provider returns a set of routes with the minimum cost that satisfy all ride requests under some spatio-temporal constraints. A survey [13] summarized the early studies on DARP. In general, DARP is NP-hard and only instances with a small number of vehicles and requests can be solved optimally, and the approaches are often based on integer programming techniques [14]. These approaches are usually implemented in two phases: the first phase is to partition the requests and obtain an initial schedule, and the second phase is to improve the solution by local search.

2.2 Dynamic Ridesharing

Motivated by the recent mobile technologies, dynamic ridesharing services have been drawing increasing attention from both industry and academia [3–5,15]. In dynamic ridesharing systems, riders and drivers continuously enter and leave the system. Dynamic ridesharing algorithms match up them in real time. Generally, the existing works can be divided into to *centralized* ridesharing and *distributed* ridesharing.

In a centralized dynamic ridesharing system, a central service provider performs all the necessary operations to match riders to drivers. Agatz et al. [16] surveyed optimization techniques for centralized dynamic ridesharing services in which different optimization objectives (e.g., minimizing system-wide overall travel distance or travel time) and spatio-temporal constraints (with desired departure/arrival time or spatial proximity constraints) are considered. Rigby et al. [17] proposed an opportunistic user interface to support centralized ridesharing planning as well as preserving location privacy. Huang et al. [4]

formulated a centralized real-time ridesharing problem with service guarantee, and proposed a novel kinetic tree based solution.

The drawback of the centralized ridesharing is the lack of scalability. To address this issue, distributed ridesharing solutions have been developed. d'Orey et al. [18] modeled a dynamic and distributed taxi-sharing system and proposed an algorithm based on peer-to-peer communication and distributed coordination. Zhao et al. [15] proposed a distributed ridesharing service based on a geometry matching algorithm that shortens waiting time for riders and avoids traffic jams.

Note that existing proposals for dynamic ridesharing do not consider social connections among riders and drivers.

2.3 Trust-Conscious Ridesharing

A few recent studies addressed the trust issue in ridesharing [2, 9, 19, 20]. Alternative approaches include adopting reputation-based systems and profile check by associating with social networks such as Facebook [9]. Cici et al. [2] proposed to assign participants who are friends or friends of friends into one group in order to gain potential benefits of ridesharing. Li et al. [19] employed k-core as the primary social model and proposed algorithms to solve social-aware ridesharing group queries. Unlike our study presented in this paper, the ridesharing model used in [19] is slugging and cannot adapt to the dynamic environment.

Table 1. Existing studies on ridesharing

Existing work	Characteristics		
	Social factor	Optimization objective	Real-time
T-share [5]	no	Travel distance	yes
Kinetic tree [4]	no	Trip cost	yes
SaRG [19]	yes	Travel cost of the group	no
TkSaTR [this paper]	yes	Ranking function	yes

Table 1 summarizes the characteristics of the most typical existing proposals for ridesharing. In this paper, we are interested in real-time taxi ridesharing services (TkSaTR) that consider the social factor and optimize the mix of spatial and social factors.

3 Problem Formulation

3.1 Definitions

In our setting, a road network is viewed as a time-dependent graph $G(N, E, W)$. Specifically, N is the set of nodes each representing a road intersection or a terminal point, and $E \in N \times N$ is the set of network edges each connecting two nodes. Representing a road segment from node n_i to n_j, edge $e_{i,j} \in E$ is associated with a time-dependent weight function $W_e(t)$ that indicates the travel cost along the edge $e_{i,j}$. In particular, $W_e(t)$ specifies how much time it takes to travel from n_i to n_j if a vehicle departs from n_i at time t.

To ease the explanation, we represent the road network as an undirected graph in the examples in this paper. Nevertheless, the algorithms we propose can handle the road network as a directed graph.

On the other hand, we model a social network as an undirected graph $G_S = (V_s, E_s)$, where a node $v_i \in V_s$ represents a user and an edge $(v_i, v_j) \in E_s$ indicates the social connection between two users v_i and $v_j \in V_s$.

Definition 1 *(**Trip Request**). A trip request is denoted by $tr = (t, o, d, pw, dt)$ where t indicates when tr is submitted, o and d represent the origin and destination, respectively, pw represents the time window when the rider wants to be picked up, and dt represents the latest drop off time.*

For a time window pw, we use $pw.e$ and $pw.l$ to denote its earliest and latest time, respectively. In practice, a rider may only need to specify $tr.d$ and $tr.dt$. The ridesharing service can automatically record $tr.t$ and $tr.o$ (if the rider is GPS-enabled). We may also set $tr.pw.e$ to $tr.t$ and $tr.pw.l$ to a time later than $tr.pw.e$ by a default time period, e.g., 5 min.

Definition 2 *(**Schedule**). A schedule s of n trip requests $tr_1, tr_2, ..., tr_n$ is a sequence of origins and destinations such that for each request tr_i, either (i) both $tr_i.o$ and $tr_i.d$ exist in the sequence and $tr_i.o$ precedes $tr_i.d$, or (ii) only $tr_i.d$ exists in the sequence.*

Each taxi in operation is associated with a schedule that changes dynamically over time. For example, if two trip requests tr_1 and tr_2 are assigned to a taxi, the initial schedule can be $(tr_1.o, tr_2.o, tr_2.d, tr_1.d)$. After the taxi picks the rider at $tr_1.o$, its associated schedule is updated to $(tr_2.o, tr_2.d, tr_1.d)$ accordingly. Each taxi continuously maintains its geographic location $T.l$, the number of on-board riders $T.n$, and its schedule $T.s$.

Definition 3 *(**Satisfaction**). Given a taxi T and a trip request tr, T satisfies tr if and only if:*

(1) $T.n$ is less than the capacity of T;
(2) T is able to pick up the rider of tr at $tr.o$ on or before $pw.l$ and drop off the rider at $tr.d$ no later than $tr.dt$;
(3) T can serve all riders currently in $T.s$ without violating any time constraints imposed by their trip requests.

It is noteworthy that a taxi T does not satisfy a request tr if T cannot arrive at $tr.o$ during time window $tr.pw$.

When a trip request tr comes into the ridesharing service, there can be many taxis that satisfy tr. To this end, we employ a ranking function $f(tr, T_i)$ that scores a taxi T_i with respect to request tr by considering (i) the travel distance to transport the new rider from $tr.o$ to $tr.d$, and (ii) the social connectivity between the new rider and existing riders in $T_i.s$. The former considers the *spatial* aspect, whereas the latter concerns the *social* aspect. The ranking function $f(tr, T_i)$ can be any monotonic aggregate function that combines social score S_i and spatial

score D_i for taxi T_i. We detail scores S_i and D_i in Sect. 3.2. We give two types of ranking functions:

$$f(tr, T_i) = \omega \cdot S_i + (1 - \omega) \cdot D_i \tag{1}$$

$$f(tr, T_i) = S_i \times D_i \tag{2}$$

The top-k taxis selection algorithms proposed in this paper are independent of the concrete definition of $f(tr, T_i)$. In the rest of this paper, we use Eq. 1 as the ranking function. In Eq. 1, parameter $\omega \in [0, 1]$ specifies the relative importance of the social and spatial factors. When $\omega > 0.5$, the social cohesion of the ride-sharing group is more important than the travel distance.

Based on the definitions above, we formally define our research problem:

Problem 1 (**Top-k Social-aware Taxi Ridesharing Query**). Given a set of taxis each having its current schedule of riders, a road network $G(N, E, W)$, a social network $G_S = (V_s, E_s)$, and a trip request tr, a top-k social-aware taxi ridesharing query (TkSaTR query for short) returns the top-k taxis satisfying tr with the highest score calculated by the ranking function $f(tr, T_i)$.

In this paper, we study the online social-aware ridesharing problem, i.e., the system does not know the information of upcoming trip requests and a submitted trip request needs to get response from the system in real time. The performance of an online algorithm is usually analyzed by *competitive ratio* [21]. An algorithm $\mathcal{A}$ is called c-competitive for a constant $c > 0$, if and only if, for any input $\mathcal{I}$ the result of $\mathcal{A}(\mathcal{I})$ is at most c times worst than the globally optimal solution. We show that no online algorithm can achieve a good competitive ratio for our social-aware rideahring problem.

Theorem 1. *There is no deterministic online algorithm for the social-aware ridesharing problem that is c-competitive ($c > 0$).*

Proof. Suppose an algorithm $\mathcal{A}$ is c-competitive, then the output of $\mathcal{A}$ must be at most c times worst than the optimal solution for every input. Consequently, to show $\mathcal{A}$ is not c-competitive, we only need to find one input that $\mathcal{A}$ cannot provide the solution no worse than c times to the optimal solution.

We assume that an adversary knows every decision $\mathcal{A}$ makes and can submit trip requests as $\mathcal{A}$'s input. For simplicity, we assume there is only one taxi locating at point $(0, 0)$. We also assume that the earliest departure time is equal to the trip request submission time and the latest departure time is w later than the earliest departure time (i.e., the maximum waiting time is w). Initially, there are two trip requests tr_1 and tr_2 with origins $(w, 0)$ and $(-w, 0)$ respectively. $\mathcal{A}$ can make three possible choices for the taxi: (1) move towards tr_1, (2) move towards tr_2 and (3) stay still. If choice 1 is selected, then the adversary would submit n more trip requests at $(-w - 1, 0)$ at time $t = 1$ and their destinations are similar to tr_2's. These n riders and tr_2 are friends with each other. Under this circumstance, the global optimal solution can serve $n + 1$ trip requests and riders' social cohesion is the highest (every rider knows each other), while $\mathcal{A}$ can

at most complete one trip request. Similar arguments can be made if choice 2 or choice 3 is selected. By submitting more trip requests in a similar manner, the adversary can make $\mathcal{A}$'s solution unboundedly worse than the global optimal solution. Therefore, the assumption of $\mathcal{A}$ is c-competitive is invalid and the proof completes. □

3.2 Spatial and Social Scores

For the spatial score, suppose $T_i.s'$ is the schedule after the new request tr is assigned to taxi T_i. We use the following equation to capture the average detour of the riders who are assigned to T_i.

$$D_i = \frac{\sum_{tr_i \in T_i.s'} t_f(tr_i.o, tr_i.d)}{\sum_{tr_i \in T_i.s'} t_{share}(tr_i.o, tr_i.d)} \tag{3}$$

For a trip request tr_i already in T_i's schedule, $t_f(tr_i.o, tr_i.d)$ is the travel time along the fastest path between tr_i's origin and destination (i.e., the travel time without using ridesharing), and t_{share} is the travel time from tr_i's origin to its destination through the ridesharing enabled by T_i.

Let R_i be the union of the user issuing the new trip request tr and all those involved in taxi T_i's current schedule. Let $sd(u_j, u_k)$ represent the social distance, i.e., the number of hops, between users u_j and u_k in a social network $G_S = (V_s, E_s)$. As an extreme case, if there is no path between nodes u_j and u_k, $sd(u_j, u_k)$ will be $dia(G_S)+1$ where $dia(G_S)$ is the diameter of G_S. Accordingly, we define the social score as $S_i = \frac{|R_i| \cdot (|R_i|-1)}{\sum_{j=1}^{|R_i|} \sum_{k=1}^{|R_i|} sd(u_j, u_k)}$. It is normalized to the range of (0, 1].

3.3 Solution Overview

A brute-force approach for a TkSaTR query enumerates all the taxis to find the top-k taxis. Given a trip request tr, we check the taxis one by one and retrieve those as the candidates that can reach $tr.o$ before time $tr.pw.l$. Subsequently, for each candidate taxi T_i's schedule $T_i.s$, we enumerate all the permutations of inserting $tr.o$ and $tr.d$ into $T_i.s$, and find those taxis that satisfy tr. These taxis are called *feasible taxis*. Finally, we calculate the ranking score for each feasible taxi and return the top-k taxis with the highest ranking scores. If there is no feasible taxis with respect to a request tr, the system may suggest the user to modify the trip request or resubmit it later. Apparently, this brute-force approach is inefficient as it does not identify and rule out unpromising taxis aggressively.

We propose an efficient framework to solve the problem, as illustrated in Fig. 1. It consists of two major modules: Candidates Searching to find candidate taxis for a new trip request and Optimal Scheduling to add the request to candidate taxis' schedules and return the top-k taxis. We design two methods for selecting candidate taxis. The grid-based method rules out unpromising taxis

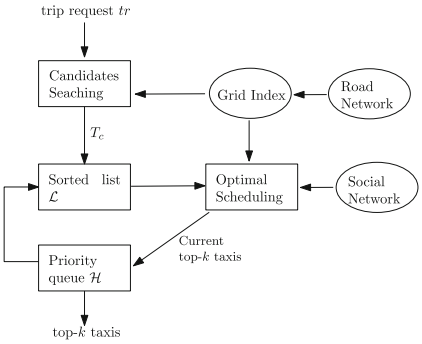


Fig. 1. System framework

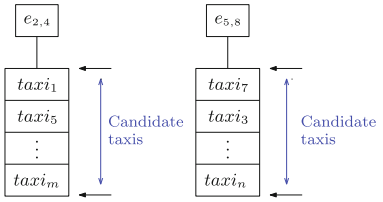


Fig. 2. Selected candidate taxis

using a grid index and returns a relatively larger candidate set, whereas the edge-based method expands the selection through the road network and obtains a smaller candidate set. For the scheduling module, we derive travel time based bounds to rule out unqualified cases (taxis and schedules) quickly. We also design algorithms to insert $tr.o$ and $tr.d$ into $T_i.s$ efficiently without violating the relevant time constraints. We only calculate the concrete ranking score for feasible and most promising taxis.

4 Candidate Taxis Searching

The candidate taxis searching process is intended to find a candidate set of taxis that are likely to satisfy a given trip request tr. In this section, we propose two candidate searching methods. The *Edge-based Candidates Selection* searches for candidate taxis by expanding from tr's origin location to reachable edges in the road network. The *Grid-based Candidates Selection* utilizes a grid index to prune unpromising taxis that are too far away from tr's origin location.

4.1 Edge-Based Candidates Selection

In this method, we maintain a list $e.L_t$ for each road network edge e. The list contains the IDs of taxis that are currently located on edge e. Taxis are appended to the list when it enters e, and a taxi is removed from the list when it leaves e. As a result, all IDs in the list $e.L_t$ are sorted in ascending order of the taxis' entering time.

An example is shown in Fig. 4. Suppose that a trip request tr arrives at time t_{cur} with its origin location at n_4. Any taxi that is currently on edges from which n_4 is reachable before $tr.pw.l$ is retrieved as a candidate taxi. This time constraint is captured in Eq. 4, where n_x indicates a taxi's current location.

$$t_{cur} + t(n_x, n_4) \leq tr.pw.l \tag{4}$$

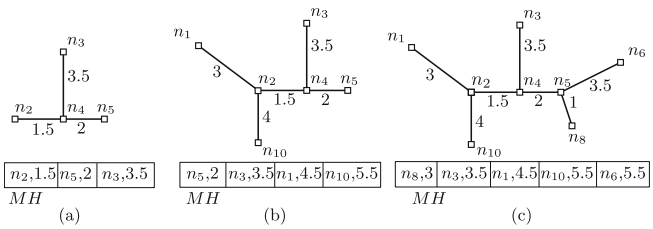


Fig. 3. Searching candidate taxis

Suppose that $tr.pw.l = tr_{cur} + 2$, which means $t(n_x, n_4)$ must less be than or equal to 2. The edge-based selection expands the search from n_4 to other nodes in the ascending order of their travel time to n_4. The expansion stops once it encounters a node n_x that fails to satisfy Eq. 4. To enhance the performance, a min-heap MH is used to store the nodes visited so far, using the minimum travel time to n_4 as the priority. Initially, n_4's adjacent nodes n_2, n_3, n_5 are en-heaped into MH. This selection stops when the de-heaped node fails to satisfy Eq. 4. As node n_2 has the minimum travel time to n_4, it is de-heaped and processed first. As it satisfies Eq. 4, the taxis in $e_{2,4}.L_t$ are retrieved as candidate taxis as shown in Fig. 2. Subsequently, nodes n_1 and n_{10} are en-heaped with their travel time to n_4, as shown in Fig. 3(b). Furthermore, n_5 is de-heaped and n_8 and n_6 are en-heaped as shown in Fig. 3(c). Next, n_8 is de-heaped as it has the minimum travel time to n_4 in MH. As $t(n_8, n_4)$ is 2, Eq. 4 is violated and the selection process stops.

4.2 Grid-Based Candidates Selection

In the grid-based candidates selection, we utilize a grid index to maintain the information and speed up the candidates searching process. The whole road network is divided by a grid. For each grid cell g_i, the information about the road network and taxis is stored in two tables: the *edge table* T_{edge} and the *node table* T_{node}.

Specifically, table T_{edge} stores the following information of each edge e within cell $g_{i,j}$: (1) the nodes n_i and n_j connecting e (if n_i is in $g_{i,j}$), (2) the grid cell in which n_j is located, (3) the travel time function $W_e(t)$, and (4) the ID list L_t of the taxis currently on e. All taxi IDs are sorted in ascending order according to their entering time to e. A taxi T_i is removed from $e.L_t$ when it leaves e. The travel time function $W_e(t)$ implies the travel time on edge e for the current timestamp t. On the other hand, table T_{node} maintains the coordinate (x, y) of each node n_i in $g_{i,j}$.

To illustrate the grid index, a road network with 17 edges and 17 nodes is shown in Fig. 4. The whole road network is divided into 4×4 grid cells, from $g_{0,0}$ to $g_{3,3}$. Take $g_{1,2}$, a gray rectangle, as an example. The contents of T_{edge} and T_{node} for $g_{1,2}$ are shown in Table 2 (the taxi lists ta of edges are omitted due to space limit). Note that only the cells enclosing the nodes of an edge need to

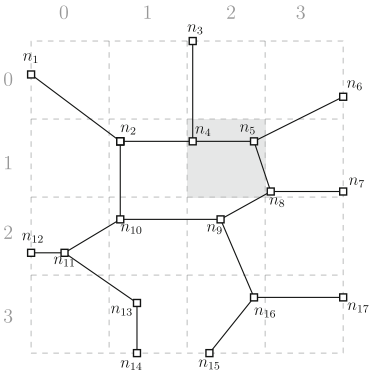


Fig. 4. A road network

Table 2. Information of $g_{1,2}$

T_{edge}

e	n_i	n_j	$T(e)$	$g_{m,n}$
$e_{4,2}$	n_4	n_2	1.5	$g_{1,1}$
$e_{4,3}$	n_4	n_3	3	$g_{0,2}$
$e_{4,5}$	n_4	n_5	2	$g_{1,2}$
$e_{5,6}$	n_5	n_6	3.5	$g_{0,3}$
$e_{5,8}$	n_5	n_8	2	$g_{1,3}$

T_{node}

n	(x, y)
n_4	(30,52)
n_5	(30,73)

maintain the relevant T_{edge} and T_{node}. For example, the information about edge $e_{1,2}$ is found in cell $g_{0,0}$ or $g_{1,1}$, but not in cell $g_{1,0}$.

With these tables, the grid-based candidates selection process is straightforward. After receiving a trip request tr whose origin is in grid cell $g_{i,j}$, the system first searches for the candidate taxis which are likely to satisfy tr. The taxis within $\left\lceil \frac{(tr.pw.l - tr.t) \cdot D}{cs} \right\rceil$ cells are the candidates as only these taxis may be able to satisfy the pickup time constraint. Here, D is the maximum velocity of a taxi and cs is the grid cell size. Any taxi beyond these cells is too far away to reach tr's origin location before tr's latest pickup time.

5 Taxi Scheduling and Top-k Taxi Selection

Given the set T_C of candidate taxis with respect of a trip request tr, the purpose of the taxi scheduling process is to find the top-k taxis satisfying tr with the highest score calculated by the ranking function $f(tr, T_i)$. We start with describing the overall procedure of top-k taxis selection.

5.1 Overall Procedure of Top-k Taxis Selection

After the candidate taxis are obtained in set T_C, we calculate the social score for each taxi in T_C and sort all candidate taxis as follows. For each candidate taxi $T_i \in T_C$, we calculate its *ranking score upper bound* as $RSUB_i = \omega \cdot S_i + (1 - \omega) \cdot D_i^+$ where S_i is T_i's social score and D_i^+ is the upper bound of T_i's spatial score D_i, both with respect to the new request tr. It is apparent that $RSUB_i \geq f(tr, T_i) = \omega \cdot S_i + (1 - \omega) \cdot D_i$ for $\omega \in (0, 1)$ (Eq. 1). We elaborate on how to derive D_i^+ in Sect. 5.2. All candidate taxis are put in a sorted list $\mathcal{L}$ in descending order of their $RSUB_i$ values.

Subsequently, we sequentially process each candidate taxi T_i in $\mathcal{L}$. In particular, we find for T_i the optimal schedule that satisfies the request tr and yields

the optimal (largest) spatial score. This will be detailed in Sect. 5.4. Afterwards, we calculate the final ranking score for taxi T_i using $f(tr, T_i)$ and push T_i into a priority queue $\mathcal{H}$ that uses the ranking score $f(tr, T_i)$ as the priority. The overall procedure stops when $\mathcal{H}$ contains at least k taxis and the current candidate taxi T_i from $\mathcal{L}$ to process has an $RSUB_i$ value no greater than the k-th final ranking score in $\mathcal{H}$. In such a case, all remaining candidates in T_i cannot have higher ranking scores than the k-th taxi in $\mathcal{H}$. As a result, the top-k taxis are already found in $\mathcal{H}$.

5.2 Spatial Score Upper Bounds

In order to derive the upper bound for the spatial score defined in Eq. 3, we estimate the lower bound for $t_{share}(tr_j.o, tr_j.d)$ used in the denominator of Eq. 3. To ease the presentation, let s be $tr_j.o$ and e be $tr_j.d$, and $t(s, e)$ be $t_{share}(tr_j.o, tr_j.d)$. We derive two lower bounds for $t(s, e)$.

A straightforward way is to consider the Euclidean distance from s to e, denoted as $d_{Euc}(s, e)$ instead of the complex road network distance. Suppose that we know the maximum speed v_{max} a taxi can travel at in the road network. Then we have

Lemma 1 ***(Euclidean Travel Time Lower Bound).*** $TTLB_{Euc}(s, e) = \frac{d_{Euc}(s,e)}{v_{max}} \leq t(s, e)$.

An alternative is to use the grid index and derive a lower bound accordingly. For each grid cell, we identify its boundary node that is immediately connected to some node in a different cell. Any path linking a node in a cell g_i with some node in another cell g_j must go through at least two boundary nodes in g_i and g_j, respectively. We derive the *Grid Travel Time Lower Bound* based on this observation. For each pair of cells g_i and g_j, we pre-compute the fastest path distance from each boundary node in g_i to each boundary node in g_j, and store the smallest one as $sgd_{i,j}$, i.e., $sgd_{i,j} = \min_{b_i \in g_i, b_j \in g_j} t(b_i, b_j)$. In each cell g_i, we pre-compute and store the fastest path distance from each node to its nearest boundary node b_{ni}. Then we have

Lemma 2 ***(Grid Travel Time Lower Bound)***
$TTLB_{grid}(s, e) = t(s, b_{ni}) + sgd_{i,j} + t(b_{nj}, e) \leq t(s, e)$.

Proof. For any path from node s in grid cell g_i to node e in grid cell g_j, it consists of three parts: (1) the path from s to some boundary node b'_{ni} in g_i, (2) the path from b'_{ni} to some boundary node b'_{nj} in g_j, (3) the path from b'_{nj} to e.

Because $t(s, b_{ni}) \leq t(s, b'_{ni})$, $sgd_{i,j} = \min_{b_i \in g_i, b_j \in g_j} t(b_i, b_j) \leq t(b'_{ni}, b'_{nj})$, and $t(b'_{nj}, e) \leq t(b_{nj}, e)$, hence $t(s, b_{ni}) + sgd_{i,j} + t(b_{nj}, e) \leq t(s, e)$, $TTLB_{grid}(s, e)$ is the lower bound of $t(s, e)$. □

Consider the example shown in Fig. 5, where s and e are located in grid cells g_i and g_j, respectively. The closest pair of boundary nodes between g_i and g_j is b_2 and b_3. For origin node s, its nearest boundary node in g_i is

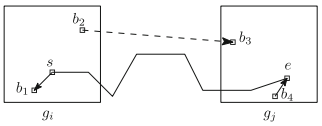


Fig. 5. Boundary node estimator

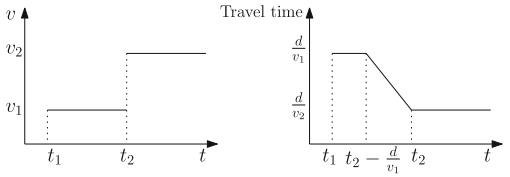


Fig. 6. Travel speed and travel time function

b_1, and destination node e's nearest boundary node in g_j is b_4. As a result, $TTLB_{grid}(s, e) = t(s, b_1) + t(b_2, b_3) + t(b_4, e) \leq t(s, e)$.

When one of the travel time lower bounds is used to replace $t_{share}(tr_j.o, tr_j.d)$ in Eq. 3, the equation gives D_i^+ that is an upper bound for the original D_i.

5.3 Time-Dependent Fastest Path Calculation

In our setting, the road network is modeled as a time-dependent graph in which the travel time may change for the same route in different periods of time. In this section, we present how to compute the fastest path from an origin s to a destination e starting at timestamp t.

Consider an edge $e_{i,j}$ with length d. Suppose that the travel speed allowed on $e_{i,j}$ is v_1 during $[t_1, t_2)$ and v_2 after t_2. Consequently, the travel time on $e_{i,j}$ (i.e., from one end to the other of $e_{i,j}$) is a continuous, piecewise-linear function of the departing ticme t from n_i [22]. Specifically, the time-dependent weight function for $t \in [t_1, t_2]$ is:

$$W_{e_{i,j}}(t) = \begin{cases} \frac{d}{v_1}, & t \in [t_1, t_2 - \frac{d}{v_1}] \\ (1 - \frac{v_1}{v_2})(t_2 - t) + \frac{d}{v_2}, & t \in [t_2 - \frac{d}{v_1}, t_2] \end{cases}$$

The relationship between speed and travel time on $e_{i,j}$ is illustrated in Fig. 6. In case that an edge contains more than two different speed patterns, $W_{e_{i,j}}(t)$ is still a continuous piecewise linear function of t with more than two linear segments.

Based on this, our procedure of fastest path calculation in a time-dependent graph is as follows. We keep a set N_E of expanded nodes and a priority queue F of frontier nodes. Initially, N_E is empty and F contains only the origin node s. Subsequently, the computation employs iterations to expand to reachable nodes from s. Each iteration chooses one node n from F, expands it by adding its unvisited neighbors to F, and then moves n to N_E. To choose the next node from F, instead of choosing n_i where the travel time from s to n_i is the smallest as in Dijkstra's algorithm, we choose the node n_j such that the travel time from s to n_j plus the estimated travel time from n_j to e is the smallest. Note that the estimate here must be a lower bound of the actual travel time in order to ensure correctness. Also, the closer the estimate is to the actual travel time, the more efficient the fastest path calculation is.

5.4 Optimal Schedule

This section introduces how to find the *optimal schedule* of a candidate taxi T given a trip request tr. The *optimal schedule* of a taxi T is the schedule with the highest spatial score after inserting tr among all the possible schedules.

To find the optimal schedule of a taxi T, intuitively we need to try all possible ways to find the way of insertion with the highest spatial score. For simplicity, here we assume the precedence relation in the current schedule remains unchanged during the scheduling process. Then inserting tr into $T.s$ can be done via two steps: (1) insert $tr.o$ into $T.s$; (2) insert $tr.d$ into $T.s$. The system checks the *feasibility* (i.e., whether T can satisfy tr or not) of all possible insertion ways for T and computes the spatial score D of those feasible ones. The spatial score of the optimal schedule of T (i.e., the highest spatial score) would be stored as the spatial score of T. For example, Fig. 7 shows one possible way to insert tr into a taxi schedule $tr_1.o \rightarrow tr_2.o \rightarrow tr_1.d \rightarrow tr_2.d$. In order to insert $tr.o$ between $tr_1.o$ and $tr_2.o$ optimally, we consider the fastest path according to current traffic condition rather than the shortest path in the road network.

For each insertion possibility, the system needs to check whether it is feasible or not. As shown in Fig. 7, to insert $tr.o$ after $tr_1.o$, the algorithm first evaluates if the taxi T is able to reach $tr.o$ before $tr.pw.l$. If not, the insertion fails. Otherwise, the system further checks whether some successor in the schedule would be delayed due to inserting $tr.o$. In $T.s$, if any other rider's time constraint is violated due to inserting $tr.o$ into $T.s$, then the insertion also fails.

If $tr.o$ is inserted successfully, the system then inserts $tr.d$ similarly. The system calculates spatial scores for all schedules which are feasible after inserting $tr.d$ and $tr.d$, the schedule with the highest spatial score will be stored as the optimal schedule of taxi T.

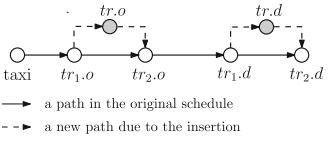


Fig. 7. One possible insertion of a ride request into a schedule

5.5 Hopping Algorithm

In order to insert a trip request tr into a schedule s and find the feasible insertion ways more efficiently, we introduce the *Hopping Algorithm* to find all the feasible insertion ways with some prune techniques. An early stop condition and two pruning conditions are derived, which are utilized in the hopping algorithm.

Early Stop Condition. Given a taxi T with schedule s and a new incoming trip request tr, if there exists some tr_x in s that satisfies both following conditions, then T definitely cannot satisfy tr: (1) $tr.dt < tr_x.pw.e$; (2) $tr.pw.e + t_f(tr.o, tr.d) > tr_x.pw.l$.

Condition (1) indicates that T must drop off tr at $tr.d$ before picking up tr_x at $tr_x.o$. In condition (2), the inequation suggests that T cannot reach $tr.d$ on or before tr_x's latest departure time even if it departs at $tr.pw.e$ and takes the fastest path from $tr.o$ to $tr.d$. Hence, tr cannot be inserted into s because inserting tr will break the time constraint of tr_x in s.

Algorithm 1. Hopping algorithm

Require: a trip request tr, a taxi T
Ensure: the set *result* of all feasible schedules after inserting tr into $T.s$

```
1: posLeft ← 0, posRight ← s.size(), result ← ∅
2: while posRight ≥ 0 do
3:     if tr.pw.e + t_f(tr.o, tr.d) > tr_{posRight+1}.dt then
4:         break                                          ▷ Pruning Condition 2
5:     else
6:         posRight − −
7: while posLeft ≤ T.s.size() do
8:     if tr_{posLeft} is a pick up point and tr.dt < tr_{posLeft}.pw.e and tr.pw.e +
       t_f(tr.o, tr.d) > tr_{posLeft}.pw.l. then
9:         return ∅                                       ▷ Early Stop Condition
10:    Schedule s ← T.s, Insert tr.o at posLeft
11:    if t_arrival > tr.pw.l then
12:        break                                          ▷ Pruning Condition 1
13:    else if insertion of tr.o succeeds then
14:        posValid ← posRight
15:        if posLeft ≥ posRiget then
16:            posValid ← posLeft
17:        for each possible insertion position i such that i > posValid in s do
18:            if insertion of tr.d at i succeeds then
19:                result ← result ∪ s, remove tr.d from s   ▷ Reset s
20:    posLeft++
21: return result
```

Taking the schedule in Fig. 7 as an example, if $tr.dt < tr_1.pw.e$ and $tr.pw.e + t_f(tr.o, tr.d) > tr_1.pw.l$, then tr cannot be inserted into this schedule no matter which positions $tr.o$ or $tr.d$ is inserted into.

Pruning Condition 1. Note that for a trip request tr, when inserting $tr.o$ between points i and j in a schedule s, if the projected arrival time along the fastest path at $tr.o$ is later than $tr.pw.l$ (i.e., $t_{arrival} > tr.pw.l$), then the insertion of all the points after i in the schedule cannot succeed either. Likewise, if the taxi cannot drop off the rider at $tr.d$ before $tr.dt$ when inserting $tr.d$ between i and j, then $tr.d$ can not be inserted after any successor of i. Based on this observation, the system is able to prune some insertion ways directly.

Pruning Condition 2. Given a trip request tr and a schedule s, when trying to insert $tr.d$ before some drop-off point $tr_x.d$ in s, if inequality $tr.pw.e + t_f(tr.o, tr.d) > tr_x.dt$ holds, then $tr.d$ cannot be inserted into any position before $tr_x.d$.

Given the Early Stop Condition and Pruning Conditions above, we propose an algorithm named Hopping Algorithm to find all the feasible schedules after inserting tr into a taxi T's schedule $T.s$. Its pseudo code is given in Algorithm 1. Basically, it utilizes two pointers, *posLeft* and *posRight*, to indicate possible

insertion positions pruned by the two pruning conditions, respectively. Given a trip request tr and a taxi T, the Hopping Algorithm returns the set *result* that contains all the feasible schedules after inserting tr into $T.s$. In $T.s$, positions before *posLeft* are the possible positions where $tr.o$ could be inserted. Likewise, positions after *posRight* are the possible positions where $tr.d$ could be inserted. Initially, *posLeft* is at the left most position of $T.s$ and *posRight* is at the right most position (lines 1). The algorithm first finds the position of *posRight* according to Pruning Condition 2 (lines 2–6). Then *posLeft* moves from left to right most position in $T.s$. If the Early Stop Condition is satisfied, then the algorithm returns $\emptyset$ which means tr cannot be inserted into $T.s$ (lines 8–9). When the Early Stop Condition is violated, the algorithm attempts to insert $tr.o$ at *posLeft*. If the projected arrival time at $tr.o$ is later than $tr.pw.l$, then $tr.o$ cannot be inserted into the current position and the positions after *posLeft* in $T.s$ according to Pruning Condition 1 (lines 10–12). If $tr.o$ is inserted successfully, then the algorithm attempts to insert $tr.d$ from the right most position to *posRight* and add the feasible schedules into *result* (lines 13–20).

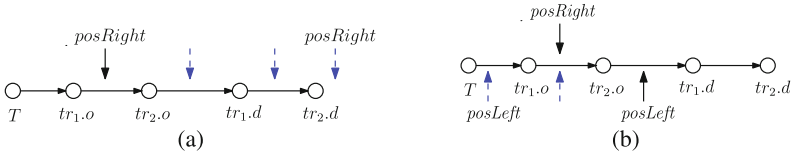


Fig. 8. An example of hopping algorithm

We take the schedule in Fig. 7 as an example to explain the procedure of the Hopping Algorithm. After initialization, the algorithm first determines the position of *posRight*. As shown in Fig. 8a, initially *posRight* stays at the right most position in $T.s$ (after $tr_2.d$) and moves towards left until Prune Condition 2 is held (shown by gray dotted arrows). Suppose that $tr.pw.e + t_f(tr.o, tr.d) > tr_2.dt$, *posRight* stops moving. This indicates that $tr.o$ cannot be inserted into any position before $tr_2.o$. Accordingly, the final position of *posRight* is shown by the black arrow in Fig. 8a. As shown in Fig. 8b, after determining the position of *posRight*, *posLeft* first stays before $tr_1.o$. As the Early Stop Condition is not satisfied and $tr.o$ can be inserted at *posLeft* successfully, the algorithm checks the positions after *posValid* (equals to *posRight* for now) to see if $tr.d$ can be inserted. The schedules with successfully inserted $tr.d$ are added into *result* set. Subsequently, *posLeft* moves towards right and stays between $tr_1.o$ and $tr_2.o$, this procedure is repeated. It is noteworthy that, if *posLeft* moves to the same position as *posRight* or the position after *posRight*, *posValid* points to the position of *posRight* since $tr.d$ cannot be inserted before $tr.o$ in $T.s$. When *posLeft* is between $tr_2.o$ and $tr_1.d$ (shown by the black arrow in Fig. 8b), the taxi cannot reach $tr.o$ before $tr.pw.l$ (Pruning Condition 1 holds) and *posLeft* stops moving. The schedules in *result* are the feasible schedules after inserting tr into $T.s$.

6 Experimental Evaluation

In this section, we experimentally evaluate the performance of our approach with the **NN** (i.e., Nearest Neighbors) approach and the **T-Share** approach [5]. We also evaluate the performance of four algorithms proposed in this paper: the brute-force approach (*BF* for short) presented in Sect. 3.3, the optimal schedule algorithm with edge-based candidate selection (*OE*), the Hopping Algorithm with edge-based candidate selection (*HE*) and the Hopping Algorithm with grid-based candidate selection (*HG*). The NN approach keeps finding the next nearest taxi. It stops either the current taxi satisfies the new trip and gets the trip, or no taxi can accept the trip and the trip is rejected by the system. All algorithms are implemented in Java, and run on a workstation with Xeon X5650 2.67 GHz CPU and 32 GB RAM.

6.1 Experimental Settings

We conduct the experiments using the real road network of New York, which contains 264,346 nodes and 366,923 edges. We evaluate the algorithms on a large scale taxi dataset containing 1,445,285 trips made by New York taxis over one month (January, 2016) [23]. We make use of the friendship network extracted from *Gowalla* [24], which consists of 196,591 nodes and 950,327 edges, to simulate users' social connections.

We classify the road segments into three categories: (1) highways, (2) roads inside downtown and (3) roads outside downtown. We assign driving velocities to the road segments as shown in Table 3.

We study the effect of the following parameters: the number of taxis, the max waiting time, the value of k and the max number of riders. For all experiments, we assume the rider always chooses the top-1 taxi. The default (bold) and other tested settings are shown in Table 4. We measure the following metrics: (1) *Taxi-sharing rate* as the percentage of trip requests participating in taxi-sharing among all trip requests, (2) *Average social score* of the trip requests which are satisfied by the system, and (3) *Query time* as the clock time for returning the top-k taxis for a given trip request.

6.2 Experimental Results

Taxi-Sharing Rate. In this set of experiments we compare the taxi-sharing rate of our approach with those of NN and T-Share. Since the four algorithms proposed in this paper perform the same in taxi-sharing rate and social score, we choose HG as the representative of our methods.

As shown in Fig. 9, our method enables more riders to share rides with others (i.e., higher taxi-sharing rate) compared to NN and T-Share under different settings, especially when the time constraint is strict or the number of taxis is large. As mentioned before, our objective is to find the best taxis that integrates both detour and riders' social cohesion. Therefore, when the number of taxis is large, our approach still works to assign a new rider to the taxi with a high

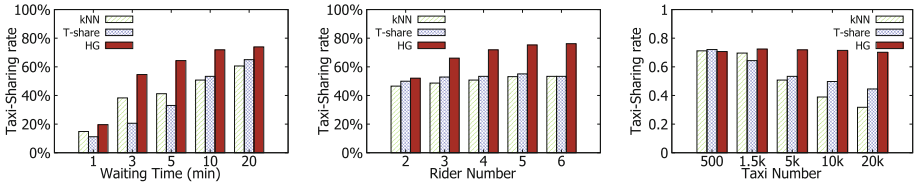
Table 3. The velocity model

Category of road	Velocity
Highway	7am–10am and 4pm–7pm: 60 km/h Otherwise 100 km/h
Roads in downtown	7am–10am and 4pm–7pm: 30 km/h Otherwise 60 km/h
Roads outside downtown	50 km/h

Table 4. Parameters settings

Parameter	Settings
# of taxis	500, 1500, **5000**, 10000, 20000
Max waiting time	1 min, 3 min, 5 min, **10 min**, 20 min
k	1, **3**, 5, 10, 20
Max riders	2, 3, **4**, 5, 6

ranking score (i.e., the average detour and the average social distance of riders in the taxi are both better) and the taxi-sharing rate remains stable. On the other hand, NN simply assigns a rider to the nearest taxi and T-Share seeks less detour for a new rider, as the number of taxis increases. Thus, both NN and T-Share approaches tend to assign a rider to an empty taxi. Hence the taxi-sharing rate of these two approaches drops dramatically.



(a) Maximum Wait Time (b) Maximum Passengers (c) Number of Taxis

Fig. 9. Comparing taxi-sharing rate of the algorithms

Social Score. As mentioned, the objective of TkSaTR query is to find the taxis that maximize the ranking score combining the spatial and social aspects. In this experiment, we compare the average social score of taxis which are chosen by the system for each trip request. Again, as our proposed algorithms yield the same social score, we take HG as the representative.

Figure 10 shows that our method achieves a higher social score (almost 40% higher) than the other approaches in almost all experimental settings. The main reason is that our method is designed to make a *social-aware* assignment, i.e., finding the top-k taxis with the highest scores considering both spatial and social factors, whereas the other approaches are not designed to achieve that goal. NN simply assigns a new trip request to the nearest feasible taxi and T-Share only considers spatial detour. Hence, the average social cohesion of riders in one taxi of these two approaches is not as good as that of our method.

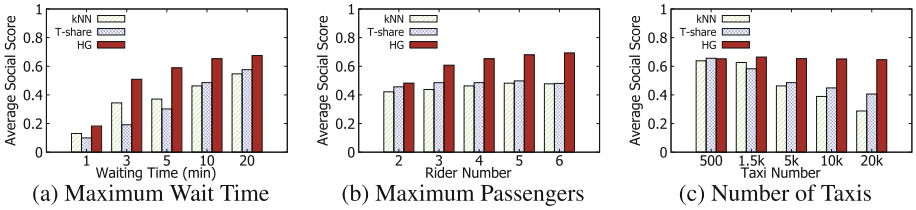


Fig. 10. Comparing average social score of the algorithms

Query Time. To evaluate the scalability of our system, we evaluate the average query processing time. Figure 11a shows that when more taxis are added, the scalability of T-Share suffers. The efficiency of our approach also decreases when the number of taxis increases, as more taxis in total tend to increase the number of candidate taxis needed to be checked in the optimal schedule model. In contrast, NN only considers the taxis near the origin of a new trip request and simply chooses the nearest feasible taxi, and therefore its performance remains stable when the number of taxis grows.

Figure 11b shows the saved distance (compared to no ridesharing) by three approaches for 1,000 queries in New York City. The saved distance increases as the number of taxis grows for all three approaches. Our method does not save as much as the other approaches since it focuses on *social-awareness* and sometimes may sacrifice spatial advantage to achieve more social cohesion. In contrast, T-Share aims to find the spatially optimal taxi (i.e., the taxi with the minimum additional travel distance with respect to a trip request). Nevertheless, our method still saves around 740 Km travel distance for 1000 queries when the taxi number is 10,000. Given that there are 13,237 taxis in New York City and there are 33,825 taxi requests per day on average (learned from the dataset), the saving achieved by our method is nevertheless over 13 million kilometers in distance per year, which equals 1 million liters of gas per year (supposing a taxi consumes 8 liter gas per 100 km) and 2.4 k ton of carbon dioxide emission per year (supposing each liter of gas consumption generates 2.3 kg carbon dioxide).

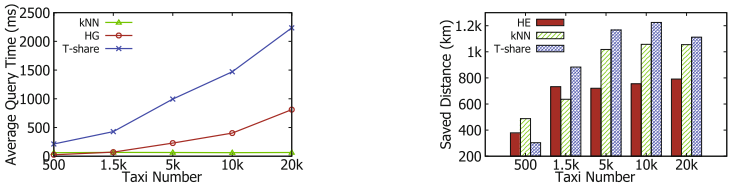
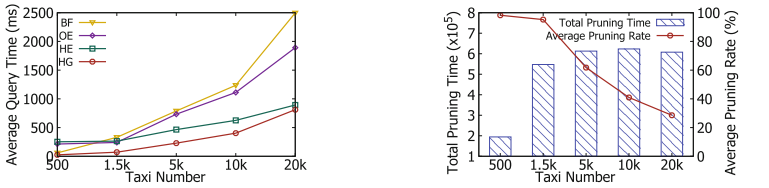


Fig. 11. Query time performance and saved distances

TkSaTR Algorithms Comparison. In this section, we experimentally evaluate the performance of the four algorithms proposed in this paper, namely BF, OE, HE, and HG.

As shown in Fig. 12a, HE and HG outperform the other algorithms with the increasing number of taxis. This is because the Hopping Algorithm used by HE and HG utilizes pruning techniques to reduce the number of possible insertion ways needed to be checked and thus mitigates the scheduling workload. Among the two candidate selection methods, the grid-based HG performs better than the edge-based HE as HG saves time in selecting candidates by utilizing the grid index. On the other hand, the scalability of BF suffers when the number of taxis is large. It enumerates all the taxis to find the top-k ones and therefore the computation load increases dramatically when encountering a large number of taxis.

To study the efficiency of the Hopping Algorithm, we evaluate the pruning times (the number of reduced feasibility checks) and the pruning rate (the percentage of reduced feasibility checks) of 1,000 queries in comparison to the approach without using Hopping Algorithm. As shown in Fig. 12b, the pruning rate is over 95% when the number of taxis is less than 1,500. Although the pruning rate drops when the number of taxis increases, the total number of checks reduced by the Hopping Algorithm is still considerably high. This suggests that the Hopping Algorithm effectively reduces the amount of computation during the optimal scheduling process.



(a) Average Query Time (b) Pruning Times and Pruning Rate

Fig. 12. Comparing performance of algorithms proposed in this paper

7 Conclusion

In this paper, we studied the problem of realtime top-k social-aware taxi ridesharing. Unlike existing studies, we introduce social-awareness into realtime ridesharing services. Our objective is to find the top-k taxis that take into account both spatial concern and social concern. With the proposed service, riders can select their preferred taxi among the top-k ones ranked in a manner that integrates spatial and social aspects. We validated our proposed solution with a large scale New York City taxi dataset. The experimental results demonstrate the effectiveness and efficiency of our proposal.

As for future work, we plan to extend the service by establishing riders' profiles that consider not only the friendships but also their interest. Another interesting direction is to process incoming trip requests within a short period of time (e.g., 5 s) in batch to improve the system throughput.

Acknowledgments. This work is supported by Hong Kong RGC grants 12200114, 12201615, 12244916 and NSFC grant 61602420.

References

1. Ma, S., Wolfson, O.: Analysis and evaluation of the slugging form of ridesharing. In: Proceedings of the 21st ACM SIGSPATIAL, pp. 64–73 (2013)
2. Cici, B., Markopoulou, A., Frias-Martinez, E., Laoutaris, N.: Assessing the potential of ride-sharing using mobile and social data: a tale of four cities. In: Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing, pp. 201–211 (2014)
3. Ma, S., Zheng, Y., Wolfson, O.: Real-time city-scale taxi ridesharing. IEEE Trans. Knowl. Data Eng. **27**(7), 1782–1795 (2015)
4. Huang, Y., Bastani, F., Jin, R., Wang, X.S.: Large scale real-time ridesharing with service guarantee on road networks. Proc. VLDB Endowment **7**(14), 2017–2028 (2014)
5. Ma, S., Zheng, Y., Wolfson, O.: T-share: a large-scale dynamic taxi ridesharing service. In: IEEE 29th International Conference on Data Engineering (ICDE), pp. 410–421 (2013)
6. Badger, E.: Slugging-the people's transit (2011)
7. Baldacci, R., Maniezzo, V., Mingozzi, A.: An exact method for the car pooling problem based on lagrangean column generation. Oper. Res. **52**(3), 422–439 (2004)
8. Calvo, R.W., de Luigi, F., Haastrup, P., Maniezzo, V.: A distributed geographic information system for the daily car pooling problem. Comput. Oper. Res. **31**(13), 2263–2278 (2004)
9. Agatz, N., Erera, A., Savelsbergh, M., Wang, X.: Sustainable passenger transportation: Dynamic ride-sharing (2010)
10. Tsubouchi, K., Hiekata, K., Yamato, H.: Scheduling algorithm for on-demand bus system. In: Information Technology: New Generations, 2009, ITNG 2009, pp. 189–194. IEEE (2009)
11. Yuan, N.J., Zheng, Y., Zhang, L., Xie, X.: T-finder: a recommender system for finding passengers and vacant taxis. IEEE TKDE **25**(10), 2390–2403 (2013)
12. Yan, S., Chen, C.Y.: An optimization model and a solution algorithm for the many-to-many car pooling problem. Ann. Oper. Res. **191**(1), 37–71 (2011)
13. Cordeau, J.F., Laporte, G.: The dial-a-ride problem: models and algorithms. Ann. Oper. Res. **153**(1), 29–46 (2007)
14. Xiang, Z., Chu, C., Chen, H.: A fast heuristic for solving a large-scale static dial-a-ride problem under complex constraints. Eur. J. Oper. Res. **174**(2), 1117–1139 (2006)
15. Zhao, W., Qin, Y., Yang, D., Zhang, L., Zhu, W.: Social group architecture based distributed ride-sharing service in vanet. Int. J. Distrib. Sens. Netw. **10**(3), 650923 (2014)
16. Agatz, N., Erera, A., Savelsbergh, M., Wang, X.: Optimization for dynamic ride-sharing: a review. Eur. J. Oper. Res. **223**(2), 295–303 (2012)

17. Rigby, M., Krüger, A., Winter, S.: An opportunistic client user interface to support centralized ride share planning. In: Proceedings of the 21st ACM SIGSPATIAL, pp. 34–43 (2013)
18. d'Orey, P.M., Fernandes, R., Ferreira, M.: Empirical evaluation of a dynamic and distributed taxi-sharing system. In: 15th International IEEE Conference on Intelligent Transportation Systems, pp. 140–146. IEEE (2012)
19. Li, Y., Chen, R., Chen, L., Xu, J.: Towards social-aware ridesharing group query services. IEEE Trans. Serv. Comput. **PP**(99), 1 (2015)
20. Bistaffa, F., Farinelli, A., Ramchurn, S.: Sharing rides with friends: a coalition formation algorithm for ridesharing (2015)
21. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM **28**(2), 202–208 (1985)
22. Kanoulas, E., Du, Y., Xia, T., Zhang, D.: Finding fastest paths on a road network with speed patterns. In: ICDE 2006, p. 10, April 2006
23. TLC: NYC TLC trip data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml
24. SNAP: Gowalla. https://snap.stanford.edu/data/loc-gowalla.html