GeoPrune: Efficiently Matching Trips in Ride-sharing Through Geometric Properties

Yixin Xu, Jianzhong Qi, Renata Borovica-Gajic, Lars Kulik
yixinx3@student.unimelb.edu.au,{jianzhong.qi,renata.borovica,lkulik}@unimelb.edu.au
The University of Melbourne, Melbourne, Australia

ABSTRACT

On-demand ride-sharing is rapidly growing. Matching trip requests to vehicles efficiently is critical for the service quality of ride-sharing. To match trip requests with vehicles, a prune-and-select scheme is commonly used. The pruning stage identifies feasible vehicles that can satisfy the trip constraints (e.g., trip time). The selection stage selects the optimal one(s) from the feasible vehicles. The pruning stage is crucial to lowering the complexity of the selection stage and to achieve efficient matching. We propose an effective and efficient pruning algorithm called GeoPrune. GeoPrune represents the time constraints of trip requests using circles and ellipses, which can be computed and updated efficiently. Experiments on real-world datasets show that GeoPrune reduces the number of vehicle candidates in nearly all cases by an order of magnitude and the update cost by two to three orders of magnitude compared to the state-of-the-art.

CCS CONCEPTS

• **Information systems** → **Location based services**; • **Applied computing** → *Transportation.*

KEYWORDS

Road networks, ride-sharing, order dispatching, real-time.

ACM Reference Format:
Yixin Xu, Jianzhong Qi, Renata Borovica-Gajic, Lars Kulik. 2020. GeoPrune: Efficiently Matching Trips in Ride-sharing Through Geometric Properties. In *32nd International Conference on Scientific and Statistical Database Management (SSDBM 2020), July 7–9, 2020, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3400903.3400912

1 INTRODUCTION

Ride-sharing is becoming a ubiquitous transportation means in our daily lives. In August 2018, there were 436,000 Uber rides and 122,000 Lyft rides per day in New York [5]. The growing number of rides calls for efficient algorithms to match numerous trip requests to optimal vehicles in real-time.

Matching trip requests to vehicles is commonly referred to as the *dynamic ride-sharing matching* problem [19, 27]. The goal is to assign each trip request to a vehicle such that a given optimization objective is achieved while satisfying the service constraints of trip requests (such as the waiting time and detour time). Various optimization goals have been studied, such as minimizing the total travel distance of vehicles [17, 19, 25, 27], maximizing the number of served requests [8], and maximizing the system profit [9, 30].

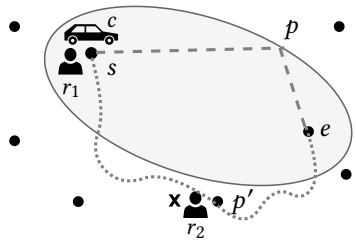


Figure 1: Illustration of our key idea

To find matches for trip requests, existing algorithms typically employ two stages: **pruning** and **selection**. The pruning stage filters out infeasible vehicles that cannot meet the service constraints of trip requests, e.g., vehicles that are too far away. From the remaining vehicles, the selection stage selects the optimal vehicles and adds the new trip requests to their routes. The computation time of the selection stage largely depends on the effectiveness of the pruning stage (i.e., the number of remaining vehicles) as it usually requires exhaustive checks on all remaining vehicles regarding the optimization goal. The pruning stage is thus crucial for both the efficiency of the selection stage and the overall matching efficiency.

We study efficient pruning of infeasible vehicles for fast matching. We focus on finding vehicles that satisfy the service constraints of trip requests rather than any particular optimization goal. Thus, our solution is generic and can be easily integrated with selection algorithms for various optimization goals. We consider essential service constraints in ride-sharing studies, the *latest arrival times* of trip requests [8–10, 17–19, 22, 25–27, 30]. Vehicles violating the constraints are infeasible matches and filtered out.

Pruning infeasible vehicles in real-time is challenging. First, ride-sharing is a highly dynamic process. New requests are arriving frequently and vehicles are moving continuously. A pruning algorithm has to not only effectively prune infeasible vehicles but also quickly update any information needed for future pruning. Second, the pruning process needs to consider the constraints of not only the new trip request but also the trip requests that are currently being served by the vehicles. Checking all these constraints poses significant challenges to the algorithm efficiency.

Existing pruning algorithms maintain dynamic indices over the road network. A simple pruning strategy is to partition the road network space into grid cells and dynamically record the grid cell

SSDBM 2020, July 7–9, 2020, Vienna, Austria

ACM ISBN 978-1-4503-8814-6/20/07…$15.00
https://doi.org/10.1145/3400903.3400912

where each vehicle resides. To match a trip request, only the vehicles in the nearby grid cells of the trip request source location need to be examined [27]. Such a strategy finds nearby vehicles but overlooks the future directions of vehicles and requests. Thus, it may return many infeasible vehicles. To obtain a higher efficiency, two approximate algorithms, *Tshare* [19] and *Xhare* [25], were proposed. Tshare precomputes pair-wise distances between grid cells and records the cells on the route of each vehicle. To match a request, Tshare checks the cells within a distance threshold of the request source/destination and retrieves vehicles passing these cells in a certain time range. Xhare, on the other hand, clusters the road network and records reachable clusters for vehicles given the time constraints. To match a request, Xhare returns all vehicles that can make a detour to the cluster where the request source/destination resides. Both algorithms may fail to find all feasible vehicles due to approximation errors such as in distance estimation, and their indices may have high storage and update costs.

To overcome the limitations above, we propose novel pruning strategies based on geometric properties of service constraints. Our strategies are built upon the following intuition. As Figure 1 shows, consider a vehicle c that has been assigned to a request r_1 with a trip from point s to point e. Vehicle c is now at s and needs to reach e within time t_1 (e.g., t_1 minutes), as constrained by r_1's latest drop-off time. To meet the time constraint t_1, vehicle c can visit a point p on its way to e only if $dist(s,p)/v_{max}+dist(p,e)/v_{max} \leq t_1$, where *dist* is the Euclidean distance between two points, and v_{max} is the maximum vehicle speed. Obviously, the vehicle may need to travel longer than the Euclidean distance as its movement is constrained by the roads. Also, it may not be able to always travel at the maximum speed. Thus, even if point p satisfies this inequality, vehicle c may still be unable to visit p. On the other hand, *if point p does not satisfy this inequality, vehicle c must not visit p*. The above inequality defines an ellipse as shown in the figure. Any point outside this ellipse violates the inequality and must not be visited by c. Thus, if there is another request r_2 from a different user at point p', we can safely prune vehicle c from consideration if p' is not in the ellipse of c. This forms the basis of our pruning strategies.

Following the idea above, we propose an efficient *geometry-based pruning* algorithm (**GeoPrune**) for ride-sharing that bounds the search space for vehicles using ellipses. We further index these ellipses using efficient data structures such as R-trees for fast search and updates. The construction of the ellipses is independent of the underlying road network and thus our algorithm is applicable to dynamic traffic-aware scenarios when vehicles may travel with different routes and speed. For every new trip request, our algorithm returns the pruning results by applying several point/range queries on the R-trees. Among the candidates, the optimal one is computed and returned with a separate selection algorithm satisfying the optimization goal. Once a trip request is assigned to a vehicle, we insert its source and destination to the vehicle route. Experimental results show that GeoPrune can prune most infeasible vehicles, which substantially reduces the computational costs of the selection stage and improves the overall matching efficiency.

The ellipse idea was explored in several matching problems [15, 21, 32]. However, their exhaustive search is inapplicable in the real-time dynamic ride-sharing settings where the relevant ellipses need to be retrieved and updated frequently and efficiently. Besides, existing algorithms represent the pruning area of every vehicle using a single ellipse to cover its entire route. Such an ellipse maybe too loose to achieve effective pruning. In contrast, our algorithm uses multiple ellipses to tightly bound the pruning area of a vehicle, which achieves more effective pruning.

Our main contributions are as follows:

- We propose novel pruning strategies to filter infeasible vehicles for trip requests. Our pruning strategies are based on geometric properties, which eliminate expensive precomputation and update costs, making them suitable for large networks and highly dynamic scenarios.
- Based on the pruning strategies, we propose an algorithm named GeoPrune that can filter out most infeasible vehicles. It significantly reduces the computational costs of the selection stage and the overall matching process. Our theoretical analysis shows that the running time of GeoPrune is $O(\sqrt{|S||C|} + |S||C|\log(|S||C|))$, where $|S|$ is the maximum number of stops of the vehicle schedules and $|C|$ is the number of vehicles. GeoPrune takes $O(|S|\log^2(|S||C|))$ time to update the states for a newly assigned trip request. During every time slot, GeoPrune takes $O(|S|\log(|S||C|)+|C|\log^2|C|)$ time to update for moving vehicles.
- Experiments on real datasets confirm the effectiveness and efficiency of our algorithm. Comparing with the state-of-the-art, it reduces the number of potential vehicles in nearly all cases by an order of magnitude and the update time by two to three orders of magnitude.

2 PRELIMINARIES

We first present basic concepts and define our ride-sharing matching problem. Table 1 summarizes the frequently used symbols.

2.1 Definitions

We consider ride-sharing on a road network that is represented as a directed graph $G = \langle N, E\rangle$, where N is a set of vertices and E is a set of edges. Each edge $e(n_i, n_j)$ is associated with weight $d(n_i, n_j)$ indicating the travel distance between vertices n_i and n_j. We denote the estimated travel time between n_i and n_j as $t(n_i, n_j)$ (which may be calculated based on their road network distance or fetched from a navigation service such as Google Maps).

Trip request. A *trip request* $r_i = \langle t, s, e, w, \epsilon, \eta\rangle$ consists of six elements: the issue time t, the source location s, the destination location e, the maximum waiting time w, the maximum detour ratio ϵ, and the number of passengers η. A set of trip requests is represented as $R = \{r_1, r_2, ..., r_n\}$.

For a trip request r_i, the issue time $r_i.t$ records the time when the trip request is sent. The maximum waiting time $r_i.w$ limits the *latest pickup time* of the request to be $r_i.lp = r_i.t + r_i.w$. The maximum detour ratio $r_i.\epsilon$ limits the extra detour time of the request. Together with the maximum waiting time, it constraints the *latest drop-off time* of the request to be $r_i.ld = r_i.t + r_i.w + t(s,e)\times(1+\epsilon)$. Alternatively, a request can directly set the latest pickup and drop-off times or simply set the latest drop-off time and the latest pickup time is then calculated as $r_i.lp = r_i.ld - t(s,e) - r_i.t$. The difference between the latest drop-off time and the issue time, i.e., $r_i.ld - r_i.t$, is its maximum allowed travel time.

Table 1: Frequently Used Symbols

Notation	Description
$G = \langle N, E \rangle$	a road network with vertex (edge) set N (E)
$t(n_i, n_j)$	the estimated shortest travel time between vertices n_i and n_j
$R = \{r_i\}$	a set of trip requests
$C = \{c_j\}$	a set of vehicles
$r_i = \langle t, s, e, w, \epsilon, \eta \rangle$	a trip request issued at time t with source s, destination e, maximum waiting time w, maximum detour ratio ϵ and η passengers
$r_i.lp, r_i.ld$	the latest pickup and drop-off times of r_i
$r_i.wc, r_i.rd$	the waiting circle and the detour ellipse of r_i
$c_j = \langle l, S, u, v \rangle$	a vehicle at l with planned trip schedule S, capacity u and traveling speed v
(p^{k-1}, p^k)	the segment between p^{k-1} and p^k
$vd[k]$	the detour ellipse of (p^{k-1}, p^k)

Example 2.1. *Assume two trip requests $r_1 = \langle$9:00 am, s_1, e_1, 5 min, 0.2, 1$\rangle$ and $r_2 = \langle$9:07 am, s_2, e_2, 5 min, 0.2, 1$\rangle$ in Figure 2. The shortest travel times from s_1 to e_1 and from s_2 to e_2, i.e., $t(s_1, e_1)$ and $t(s_2, e_2)$, are both 15 min. Then, the time constraints of r_1 and r_2 are: $r_1.lp$=9:00 am+5 min=9:05 am, $r_2.lp$=9:07 am+5 min=9:12 am, $r_1.ld$=9:05 am+15 min×1.2=9:23 am, $r_2.ld$=9:12 am+15 min×1.2=9:30 am.*

Vehicle. A *vehicle* c_i is represented as $c_i = \langle l, S, u, v \rangle$, where l denotes the location of the vehicle, S represents the *trip schedule* of the vehicle (detailed later), u is the vehicle capacity, and v is the travel speed. We use $C = \{c_1, ..., c_n\}$ denotes a set of vehicles.

We track the occupancy status of the vehicles [13]. A vehicle is *empty* if it has not been assigned to any trip requests. Otherwise, the vehicle is *non-empty* and needs to follow their trip schedules.

2.2 Vehicle schedule

Trip schedule. The trip schedule of a vehicle c_i, $c_i.S = \{p^0, p^1, ..., p^m\}$, is a sequence of source or destination locations (points on the road network) of trip requests, except for p^0 that records the current location of the vehicle, i.e., $p^0 = c_i.l$. We call a source or destination location on a trip schedule a *stop*, and the path between every two adjacent stops p^{k-1} and p^k a *segment*, denoted as (p^{k-1}, p^k).

Example 2.2. *Figure 2 shows an example trip schedule. The current time is 9:00 am and the vehicle is at l. Two trip requests (r_1, r_2) are assigned to the vehicle and the vehicle schedule is $(l, r_1.s, r_2.s, r_1.e, r_2.e)$.*

Trip schedule recorder. We follow a previous study [27] and record the *earliest estimated arrival time*, *latest arrival time*, and *slack time* of $c_i.S$ with three arrays $arr[]$, $ddl[]$, and $slk[]$:

(1) Earliest estimated arrival time $arr[k]$ records the estimated arrival time to stop p^k via the trip schedule.

(2) Latest arrival time $ddl[k]$ records the latest acceptable arrival time at stop p^k. If p^k is the pickup point of a request r_j, $ddl[k]$ is the latest pickup time of r_j, $ddl[k] = r_j.lp$. If p^k is the drop-off point of r_j, $ddl[k]$ is the latest drop-off time of r_j, $ddl[k] = r_j.ld$.

(3) Slack time $slk[k]$ records the maximum extra travel time allowed between (p^{k-1}, p^k) to satisfy the latest arrival time of p^k and all stops scheduled after p^k. For stop p^i, it only allows

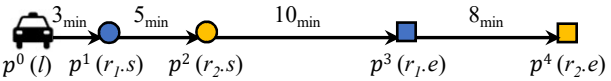


Figure 2: A vehicle schedule example at 9:00 am.

Table 2: Recorded Data for the Trip Schedule in Figure 2.

p^k	$arr[k]$	$ddl[k]$	$ddl[k] - arr[k]$	$slk[k]$
p^1	9:03 am	9:05 am	2 min	2 min
p^2	9:08 am	9:12 am	4 min	4 min
p^3	9:18 am	9:23 am	5 min	4 min
p^4	9:26 am	9:30 am	4 min	4 min

$ddl[i] - arr[i]$ detour time to ensure its latest arrival time. A detour between p^{k-1} and p^k will not only affect the arrival time of p^k but also that of all stops scheduled after p^k. Thus, a detour between p^{k-1} and p^k must guarantee the latest arrival time of p^k and all stops scheduled after p^k, i.e., $slk[k] = min\{ddl[i] - arr[i]\}, i = k, ..., m$. $slk[k]$ can be calculated by referring to $slk[k+1]$, i.e., $slk[k] = min\{(ddl[k] - arr[k]), slk[k+1]\}$. The *maximum allowed travel time* between (p^{k-1}, p^k) is $arr[k] - arr[k-1] + slk[k]$.

Example 2.3. *The arrays of the trip schedule in Figure 2 are shown in Table 2. The earliest estimated arrival time of the stops is computed based on the arrival time of previous stops and the shortest travel time between stops, e.g., $arr[1]$=9:00 am+3 min= 9:03 min, $arr[2]$ =9:03 am+5 min= 9:08 min. The latest arrival time of the stops is determined by the corresponding trip requests, e.g., the latest arrival time of p^1 is the latest pickup time of r_1, i.e., $ddl[1] = r_1.lp$=9:05 am. $ddl[k] - arr[k]$ represents the allowed detour time before visiting p^k to ensure $ddl[k]$, e.g., p^1 allows 9:05 am-9:03 am=2 mins detour before it and p^2 allows 9:12 am-9:08 am =4 mins detour before it. $slk[k]$ records the minimum allowed detour time of p^k and all stops after p^k, e.g., a detour before p^3 will not only affect the arrival time of p^3 but also that of p^4. Thus, $slk[3] = min\{$5 min,4 min$\}$=4 min.*

Valid trip schedule. To form a *valid trip schedule*, the following trip constraints need to be satisfied:

(1) *Point order constraint:* Trip schedule $c_i.S$ must visit the pickup location $r_j.s$ before the drop-off location $r_j.e$, for any trip request r_j assigned to vehicle c_i.

(2) *Time constraint.* Trip schedule $c_i.S$ must meet the constraints for every request r_j assigned to vehicle c_i, i.e., r_j needs to be picked up before $r_j.lp$ and be dropped off before $r_j.ld$.

(3) *Capacity constraint.* At any time when c_i is traveling with trip schedule $c_i.S$, the number of passengers in the vehicle must be within the vehicle capacity.

Feasible match. Given a new trip request r_n, assigning c_i to serve r_n is *feasible* if adding r_n into the trip schedule of c_i yields a valid trip schedule. Vehicle c_i is then a *feasible vehicle* for r_n.

Similar to the previous studies [11, 27, 28], we assume that the source and the destination of the new trip request are inserted or appended to the current schedule of the matching vehicle.

2.3 Matching objective

Problem definition. Given a road network G, a set of vehicles C, a set of requests R, and an optimization objective O, we aim to match every request $r \in R$ with a feasible vehicle $c \in C$ to optimize O.

We examine a popular optimization objective, *minimizing the total increased travel distance (time)* [8, 17, 19, 25, 27]. Suppose that the total travel time of the current trip schedules of all vehicles is T, and the total travel time becomes T' after assigning vehicles in C to serve requests in R, our optimization goal O is to minimize $T' - T$.

Minimizing the total increased distance for all vehicles is NP-complete [19], and the future trip requests are unknown. A common solution is to greedily assign each trip request to an optimal vehicle [10, 19, 27, 28] ordering by their issue time. For every trip request, we assign it to a feasible vehicle such that the increased distance of the vehicle trip schedule is minimized.

2.4 Pruning and selection

We take a two-stage approach to solve the problem:

(1) **Pruning.** Given a new request r_n, the pruning stage filters out infeasible vehicles and returns a set of vehicle candidates C'.

(2) **Selection.** Given a set of vehicle candidates C', the selection stage finds the optimal feasible vehicle in C'.

In what follows, we develop algorithms for the pruning stage. Observing that empty vehicles can be pruned by applying existing spatial network algorithms [6, 23], we distinguish non-empty/empty vehicles and focus on pruning non-empty vehicles.

3 GEOMETRIC-BASED PRUNING

When a new trip request arrives, we find an optimal feasible vehicle and add the source and destination of the new trip request to the vehicle trip schedule. As discussed before, the trip schedule of the vehicle must satisfy the service constraints of all trip requests assigned to it including the new trip request. This is the basis of our pruning strategies.

There are two possibilities to add a stop to a trip schedule, either inserting it into a segment of the schedule or appending it to the end. For example, to add a new stop p to the trip schedule in Figure 2, we can either insert it to a segment to form a new schedule such as $(p^0, p, p^1, p^2, p^3, p^4)$ (we cannot insert before p^0 because p^0 is the current location of the vehicle) or append it to the end where the schedule becomes $(p^0, p^1, p^2, p^3, p^4, p)$. We say that a stop is *added* to a schedule if it is either inserted or appended to the schedule and the adding is *valid* if it still generates a valid trip schedule.

We first detail the criteria to determine whether adding the source or the destination of a new trip request is valid. These are based on constraints of the new trip and the existing trip schedule. Then, we summarize these criteria into three pruning rules.

3.1 Constraints based on existing trip requests

Given a segment (p^{k-1}, p^k), if we insert a new stop p to it, the path from p^{k-1} to p^k becomes (p^{k-1}, p, p^k). The travel time from p^{k-1} to p^k becomes $t(p^{k-1}, p) + t(p, p^k)$, which must be no larger than the maximum allowed travel time of the segment $arr[k] - arr[k-1] + slk[k]$ to satisfy the constraints of exiting trip requests.

The maximum allowed travel time limits the area that the vehicle can reach between p^{k-1} and p^k. Our key observation is that such a reachable area can be bounded using an **ellipse** $vd[k]$, and we call it the *detour ellipse* of the segment.

Definition 1. *The* detour ellipse $vd[k]$ of a segment (p^{k-1}, p^k) *is an ellipse with p^{k-1} and p^k as its two focal points, and the major axis length $vd[k].major$ equals to the maximum allowed travel time multiplied by the vehicle speed v, i.e., $vd[k].major = (arr[k] - arr[k-1] + slk[k]) \cdot v$*

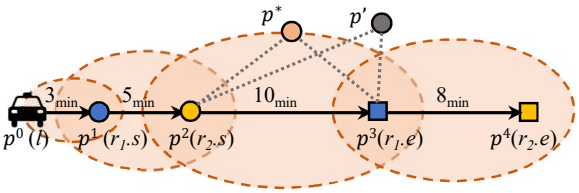


Figure 3: Detour ellipses of the trip schedule in Figure 2.

Lemma 1. *For a segment (p^{k-1}, p^k), if a point p is outside of $vd[k]$, $t(p^{k-1}, p) + t(p, p^k)$ will exceed the maximum allowed travel time. The segment is therefore invalid for inserting p.*

Proof. According to the definition of ellipses, if a point p is outside of the ellipse, the sum of the Euclidean distances $|p^{k-1}p| + |pp^k|$ must be greater than $vd[k].major$. Since any road network distance between two points is no smaller than their Euclidean distance (triangle inequality), the sum of road network distances $d(p^{k-1}, p) + d(p, p^k)$ is at least as large as $|p^{k-1}p| + |pp^k|$ and thus must also be greater than $vd[k].major$. The time required to travel such a distance thus exceeds the maximum allowed travel time and violates the latest arrival time of existing stops. □

Example 3.1. *Figure 3 shows the detour ellipses of the trip schedule illustrated in Figure 2. For segment (p^2, p^3), the slack time is 4 min and thus the maximum allowed travel time from p^2 to p^3 is 10 min + 4 min = 14 min. We make an ellipse with p^2 and p^3 as the two focal points and the major axis length being 14 min multiplied by the vehicle speed, i.e., $|p^2p^*| + |p^*p^3| = (14\,\text{min} \cdot v)$ for a point p^* on the ellipse. If a point p' is outside this ellipse, then the Euclidean distance $|p^2p'| + |p'p^3| > (14\,\text{min} \cdot v)$. The road network distance $d(p^2, p') + d(p', p^3)$ will also be greater than $(14\,\text{min} \cdot v)$ and the corresponding travel time with speed v will exceed 14 min, which violates the service constraint of exiting trip requests. Therefore, it is invalid to insert p' between (p^2, p^3).*

Lemma 1 shows that any point outside of the constructed ellipse is unreachable and thus the ellipse provides an upper bound of reachable areas. Next, we further show that the ellipse is also a lower bound of the reachable area regardless of the underlying road network, i.e., the ellipse is tight.

Lemma 2. *The detour ellipse $vd[k]$ tightly bounds the points that the vehicle can reach between segment (p^{k-1}, p^k) without violating the constraints of its existing tripe schedule.*

Proof. We prove by contradiction. Suppose that there is a smaller ellipse $vd[k]'$ with the same foci as ellipse $vd[k]$ and a major axis length of $vd[k].major - \epsilon$ ($\epsilon > 0$ is a sufficiently small value), which bounds all reachable points. This ellipse is fully enclosed by $vd[k]$. Now consider a point p^* on the boundary of $vd[k]$, which is outside $vd[k]'$ by definition. If the underlying road network happens to contain two straight routes from p^{k-1} to p^* and from p^* to p^k, which allows the vehicle to travel with the maximum speed. Then, p^* is reachable and it is outside $vd[k]'$. This contradicts the claim that $vd[k]'$ bounds all reachable points and completes the proof. □

Most existing ride-sharing matching algorithms do not consider the variations in traffic, and they assume a constant travel speed [22]. We replace the constant speed assumption with a **maximum** speed when computing the ellipses. This enables our approach to avoid false negatives if vehicles travel at varying speeds: all feasible vehicles are kept (by Lemma 1) as long as they do not exceed the maximum speed. We later show that using the maximum speed still preserves pruning efficiency. In practical implementation, we may also use different maximum speeds for different areas, e.g., in Victoria (a state in Australia), the speed limit in most built-up areas is under 60 km/h while that in rural areas is under 120 km/h.

The ellipse construction is independent of the vehicle trajectories. It only relies on the maximum allowed travel time and the endpoints of a trip segment. *Vehicles are not restricted to follow the shortest paths but are flexible to take any dynamic routes at varying speeds.* We record the ellipses of vehicles and update them only if the corresponding segments change. Specifically, when a trip request is assigned to a vehicle, we update the vehicle trip schedule and recompute the ellipses. Meanwhile, when the vehicles reach stops on their trip schedules, the corresponding segments become obsolete. We remove the ellipses of such obsolete segments.

Due to the real-time traffics and dynamic paths of vehicles, the actual arrival times at stops may be delayed and thus affect the vehicles' reachable area. $arr[]$ records the earliest arrival times and the ellipses always bound the reachable area. These allow lazy updates to vehicle ellipses when vehicles move, i.e., we do not need to recompute the ellipses when the actual arrival time is delayed.

3.2 Constraints based on the new request

Next, we analyze the service constraints of new requests.

Latest pickup time constraint. Recall that $r_n.w$ denotes the maximum waiting time to ensure the latest pickup time of the new request r_n. We define a *waiting circle* with $r_n.w$.

Definition 2. *The* waiting circle of r_n, *denoted by* $r_n.wc$, *is a circle centered at* $r_n.s$ *and with* $r_n.w \cdot v$ *as its radius.*

Lemma 3. *If it is valid to add* $r_n.s$ *after a stop* p^k *in* $c_i.S$, *then* p^k *and all stops before* p^k *must be covered by* $r_n.wc$.

Proof. The waiting circle bounds the area a vehicle can reach before picking up r_n to ensure the latest pickup time of r_n. Points outside of $r_n.wc$ have Euclidean distances (and hence network distances) to $r_n.s$ greater than $r_n.w \cdot v$. If a vehicle needs to visit a point outside of $r_n.wc$ before reaching $r_n.s$, it cannot pickup r_n before the latest pickup time $r_n.lp$. □

Example 3.2. *Figure 4 shows the waiting circle of a new request* r_n. *The source* $r_n.s$ *can only be added after the stops in the waiting circle* $r_n.wc$, *i.e.,* p^0 *or* p^1. *If the vehicle visits* p^2 *(outside of the waiting circle) before* $r_n.s$, *it will not pick up* r_n *before the latest pickup time of* r_n. *Thus, it is invalid to add* $r_n.s$ *after* p^2 *or any stops afterwards.*

Latest drop-off time constraint. Similar to the detour ellipses of segments, we define a detour ellipse for a new request r_n to ensure the latest drop-off time of r_n.

Definition 3. *The* detour ellipse $r_n.rd$ of a new trip request r_n *is an ellipse with* $r_n.s$ *and* $r_n.e$ *as the two focal points. The major axis length is the maximum allowed travel time of* r_n *multiplied by the speed* v, *i.e.,* $r_n.rd.major = (r_n.ld - r_n.t) \cdot v$.

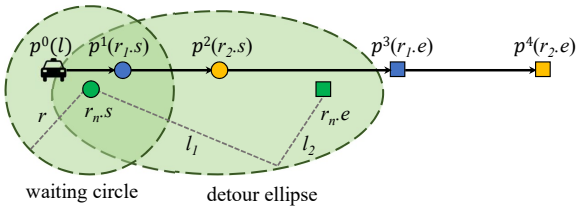
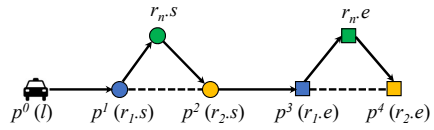
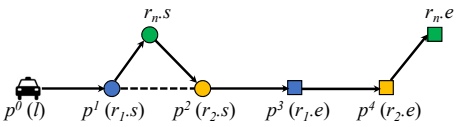


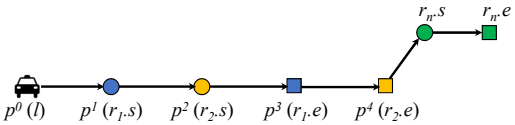
Figure 4: Waiting circle and detour ellipse of r_n, $r = r_n.w \cdot v$, $l_1 + l_2 = (r_n.ld - r_n.t) \cdot v$.



(a) insert source and insert destination.



(b) insert source and append destination.



(c) append source and append destination.

Figure 5: Cases to add a new trip request to a trip schedule

The detour ellipse of r_n restricts the area that a vehicle can visit while serving r_n. After picking up r_n (reaching $r_n.s$), if the vehicle visits any stop outside of the detour ellipse of r_n, it will not be able to reach the destination $r_n.e$ before the latest drop-off time $r_n.ld$.

Lemma 4. *Let* $r_n.s$ *be added after stop* p^s *in the trip schedule* $c_i.S$ *of a vehicle* c_i. *If it is valid to add* $r_n.e$ *after* p^k *in* $c_i.S$, *then* p^k *and all stops scheduled between* p^s *and* p^k *must be covered by* $r_n.rd$.

Example 3.3. *The detour ellipse of* r_n *is shown in Figure 4. If* $r_n.s$ *is added after* p^0, *then* $r_n.e$ *can only be added after either* p^0 *or stops inside of the detour ellipse, i.e.,* p^1 *and* p^2. *Adding* $r_n.e$ *after later stops (e.g.,* p^3*) will violate the latest drop-off time of* r_n.

3.3 Pruning Rules

There are three cases as shown in Figure 5 when adding a new trip request r_n to the trip schedule $c_i.S$ of a vehicle c_i:

(1) **insert-insert:** insert $r_n.s$ into a segment of $c_i.S$ and insert $r_n.e$ into the same or another segment of $c_i.S$.

(2) **insert-append:** insert $r_n.s$ into a segment of $c_i.S$ and append $r_n.e$ to the end of $c_i.S$

(3) **append-append:** append $r_n.s$ and $r_n.e$ to the end of $c_i.S$.

We next analyze the conditions that c_i needs to satisfy so that adding r_n to $c_i.S$ is valid for each case.

Insert-insert. Figure 5a shows the insert-insert case, where both $r_n.s$ and $r_n.e$ are inserted into some segments of the trip schedule $c_i.S$. According to Lemma 3, a segment is valid for inserting a stop

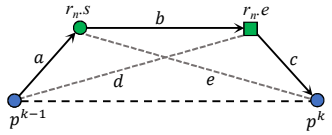


Figure 6: The special case of insert-insert.

only if the stop is inside its detour ellipse. Thus, both $r_n.s$ and $r_n.e$ must be inside the detour ellipse of at least one segment of $c_i.S$.

A special case is to insert both $r_n.s$ and $r_n.e$ to the same segment of $c_i.S$, as shown in Figure 6. In this case, both $r_n.s$ and $r_n.e$ must be inside the detour ellipse of the segment.

Lemma 5. *A segment (p^{k-1}, p^k) is valid to insert both $r_n.s$ and $r_n.e$* only if *$r_n.s$ and $r_n.e$ are both included in the detour ellipse of the segment $vd[k]$.*

Proof. We use Figure 6 to illustrate our proof. The Euclidean distances among the stops are represented by a, b, c, d, e. Suppose that (p^{k-1}, p^k) is valid to insert both $r_n.s$ and $r_n.e$, and the schedule becomes $(p^{k-1}, r_n.s, r_n.e, p^k)$ after the insertion. Traveling between (p^{k-1}, p^k) must satisfy the maximum allowed travel time constraint. Thus, $d(p^{k-1}, r_n.s) + d(r_n.s, r_n.e) + d(r_n.e, p^k) = (t(p^{k-1}, r_n.s) + t(r_n.s, r_n.e) + t(r_n.e, p^k)) \cdot v \leq (arr[k] - arr[k-1] + slk[k]) \cdot v_{max} = vd[k].major$. Since the Euclidean distance between two stops is no larger than their road network distance, $a + b + c \leq d(p^{k-1}, r_n.s) + d(r_n.s, r_n.e) + d(r_n.e, p^k) \leq vd[k].major$. According to the triangle inequality, $e < b + c$. Thus, $a + e < a + b + c \leq vd[k].major$. The Euclidean distance sum from $r_n.s$ to p^{k-1} and p^k is smaller than $vd[k].major$ and $r_n.s$ must be inside $vd[k]$. Similarly, $d < a + b$, and $d + c < a + b + c \leq vd[k].major$. $r_n.e$ must be inside $vd[k]$. □

The pruning rule for the insert-insert case is as follows:

Lemma 6. *A vehicle c_i may be matched with r_n in the insert-insert case* only if *it satisfies:*

(1) there exists a segment of $c_i.S$ with the detour ellipse that covers $r_n.s$, i.e., $r_n.s \in vd[k]$, $k = 1, ..., m$; and

(2) there exists a segment of $c_i.S$ with the detour ellipse that covers $r_n.e$, i.e., $r_n.e \in vd[k]$, $k = 1, ..., m$.

Insert-append. Figure 5b illustrates the insert-append case. According to Lemma 1, to insert $r_n.s$, there must be a segment in the trip schedule of c_i whose detour ellipse cover $r_n.s$. Meanwhile, any stop between $r_n.s$ and $r_n.e$ needs to be covered by the detour ellipse of r_n (see Lemma 4).

Checking all the stops between $r_n.s$ and $r_n.e$ against the detour ellipse of r_n is non-trivial. For fast pruning, we only check the ending stop of the current trip schedule: if the ending stop is outside of the detour ellipse of r_n, it is invalid for appending $r_n.e$. Take Figure 5b as an example. We only check if p^4 is inside the detour ellipse of r_n. This simplified rule may bring in a small number of infeasible vehicles, which will be filtered later as explained in the next paragraphs. The pruning rule for the insert-append case is:

Lemma 7. *A vehicle c_i may be matched with r_n in the insert-append case* only if *it satisfies:*

(1) there exists a segment of $c_i.S$ with the detour ellipse that covers $r_n.s$, i.e., $r_n.s \in vd[k]$, $k = 1, ..., m$; and

(2) the ending stop of the vehicle schedule, p^m, is covered by the detour ellipse of r_n, i.e., $p^m \in r_n.rd$.

Append-append. Figure 5c illustrates the append-append case, where we append both $r_n.s$ and $r_n.e$ to the end of the trip schedule. In this case, r_n will not affect any exiting stops. Only the service constraints of r_n need to be considered. No stop is scheduled between $r_n.s$ and $r_n.e$, and hence the detour constraint of r_n is satisfied already. We only need to check is the waiting time constraint of r_n. According to Lemma 3, all stops scheduled before $r_n.s$ must be covered by the waiting circle of r_n, e.g., the vehicle needs to visit p^0, p^1, p^2, p^3, p^4 before picking up $r_n.s$ in Figure 5c. Hence, all these stops should be covered by the waiting circle of r_n. Similar to the insert-append case, we only check the ending stop.

Lemma 8. *A vehicle c_i may be matched with r_n in the append-append case* only if *the ending stop of its trip schedule, p^m, is covered by the waiting circle of r_n, i.e., $p^m \in r_n.wc$.*

We omit the proof of Lemma 4, Lemma 6, Lemma 7, and Lemma 8 due to the space limitation. In our implementation, we use minimum bounding rectangles (MBRs) to represent ellipses and circles as they are easier to operate on and tightly bound the ellipses and circles.

3.4 Applying the Pruning Rules

When a new request r_n arrives, we first compute the waiting circle and the detour ellipse of r_n. Then, we compute a set of vehicle candidates that may match r_n based on Lemmas 6, 7, 8.

To facilitate the pruning, we compute sets of vehicles that:

(1) have trip schedule segments with detour ellipses that cover $r_n.s$ (for the insert-insert and insert-append cases);

(2) have trip schedule segments with detour ellipses that cover $r_n.e$ (for the insert-insert case);

(3) have the ending stop of the trip schedule covered by $r_n.wc$ (for the append-append case);

(4) have the ending stop of the trip schedule covered by $r_n.rd$ (for the insert-append case).

To find vehicles that satisfy a pruning rule, we just need to join the relevant sets of vehicles computed above. For example, vehicles that may satisfy the insert-insert case are those in both the first and the second sets above.

R-tree based pruning. We build two R-trees [16] to accelerate the computation process, although other spatial indices may also be applied. One R-tree store the detour ellipses of all segments for all vehicle trip schedules, denoted by T_{seg}; the other R-tree stores the location of the ending stops of all non-empty vehicles, denoted as T_{end}. We run four queries:

(1) $Q_1 = T_{seg}.pointQuery(r_n.s)$ is a point query that returns all segments whose detour ellipses cover $r_n.s$; each segment returned may be used to insert $r_n.s$.

(2) $Q_2 = T_{seg}.pointQuery(r_n.e)$ is a point query that returns all segments whose detour ellipses cover $r_n.e$; each segment returned may be used to insert $r_n.e$.

(3) $Q_3 = T_{end}.rangeQuery(r_n.wc)$ is a range query that returns all ending stops covered by $r_n.wc$; each ending stop returned may be used to append $r_n.s$ and $r_n.e$.

(4) $Q_4 = T_{end}.rangeQuery(r_n.rd)$ is a range query that returns all ending stops covered by $r_n.rd$; each ending stop returned may be used to append $r_n.e$.

The returned segments and ending stops are further pruned based on their time and capacity constraints. For each segment

(p^{k-1}, p^k) returned for inserting $r_n.s$ ($r_n.e$), we check whether the insertion violates the latest arrival time of p^k and $r_n.s$ ($r_n.e$). The schedule between (p^{k-1}, p^k) becomes $(p^{k-1}, r_n.s(r_n.e), p^k)$ after the insertion. For the new schedule, the arrival time of $r_n.s$ ($r_n.e$) and p^k is estimated based on the arrival time of p^{k-1} plus the travel time between them. If the earliest estimated arrival time of $r_n.s(r_n.e)$ or p^k exceeds their latest arrival time, the segment is discarded. For each ending stop (p^m) returned for appending $r_n.s$ ($r_n.e$), we estimate the arrival time of $r_n.s$ ($r_n.e$) with the appended schedule by summing up the end stop arrival time and the travel time from the end stop to $r_n.s$ ($r_n.e$). If the estimated time exceeds the latest arrival time of $r_n.s$ ($r_n.e$), we also discard the ending stop. We also check the capacity constraint for segments to insert $r_n.s$. If a segment (p^{k-1}, p^k) is returned for inserting $r_n.s$, we sum up the number of passengers carried in (p^{k-1}, p^k) and that of r_n and discard the segment if the sum exceeds the capacity.

Let the sets of vehicles corresponding to the segments and ending stops returned by the four queries above (after filtering) be O_1, O_2, O_3 and O_4, respectively. The sets of vehicles satisfying the three pruning cases are: $F_1 = O_1 \cap O_2$ (insert-insert); $F_2 = O_1 \cap O_4$ (insert-append); $F_3 = O_3$ (append-append). The union of these three sets, $F = F_1 \cup F_2 \cup F_3$, is returned as the candidate vehicles.

Processing empty vehicles. Empty vehicles do not have designated trip schedules yet. We only need to check whether they are in the waiting circle of the new request by a range query over all empty vehicles using the waiting circle as the query range.

Since our goal is to minimize the system-wide travel time, the optimal empty vehicle is the nearest one. We thus take a step further and directly compute the optimal empty vehicle with a network nearest neighbor algorithm named *IER* [23] that has been shown to be highly efficient [6] (other algorithms may also apply [24]).

4 THE GEOPRUNE ALGORITHM

Next, we describe our *pruning*, *match update*, and *move update* algorithms based on the pruning rules above.

Pruning. Algorithm 1 summarizes the pruning process. For a new request r_n, we compute its waiting circle and detour ellipse (line 1). We run four queries to compute Q_1, Q_2, Q_3, and Q_4 as described in Section 3.4 (lines 2 to 5). Each returned segment and ending stop is checked against the capacity and time constraints as described in Section 3.4 (lines 6 to 8). The vehicles of the remaining segments and ending stops are our candidates (lines 10 to 15).

Match update. If a new trip request r_n is matched with a vehicle c_i, we update the data structures as summarized in Algorithm 2. If c_i is an empty vehicle, the vehicle now becomes occupied. We remove the vehicle from an R-tree denoted by T_{ev} that stores the empty vehicles for fast nearest empty vehicle computation (lines 1 and 2). Otherwise, we first remove the segments and the ending stop of c_i from the two R-trees T_{seg} and T_{end} (lines 4 to 6). Then, we add the new trip request to the trip schedule of the matched vehicle c_i (line 7). Based on the updated vehicle schedule, we recompute the detour ellipses and insert them into T_{seg} (lines 8 to 10). The new ending stop is also inserted into T_{end} (line 11).

Move update. We also update the data structures when the vehicles move, as summarized in Algorithm 3. At every time point, we check if a vehicle has reached a stop in its trip schedule. If yes, the segments before the reached stop become obsolete and their detour ellipses are removed from T_{seq} (lines 1 to 3). When the vehicle reaches its ending stop, the vehicle becomes empty. We remove it from T_{end} and insert it into T_{ev} (lines 4 to 6).

Algorithm 1: Prune non-empty vehicles

Input: A new trip request r_n
Output: a set of possible vehicles to serve r_n
// Pruning stage
1 $r_n.wc \leftarrow$ waiting circle of r_n; $r_n.rd \leftarrow$ detour ellipse of r_n
2 $Q_1 \leftarrow T_{seg}.pointQuery(r_n.s)$
3 $Q_2 \leftarrow T_{seg}.pointQuery(r_n.e)$
4 $Q_3 \leftarrow T_{end}.rangeQuery(r_n.wc)$
5 $Q_4 \leftarrow T_{end}.rangeQuery(r_n.rd)$
6 **for** *an element in* Q_1, Q_2, Q_3, *and* Q_4 **do**
7 **if** *the time or capacity constraint is violated* **then**
8 remove the element
9 Record the corresponding vehicles of the elements in Q_1, Q_2, Q_3, Q_4 in O_1, O_2, O_3, O_4.
10 $F, F_1, F_2, F_3 \leftarrow \emptyset$
11 $F_1 \leftarrow O_1 \cap O_2$ // insert-insert case
12 $F_2 \leftarrow O_1 \cap O_4$ // insert-append case
13 $F_3 \leftarrow O_3$ // append-append case
14 $F \leftarrow F_1 \cup F_2 \cup F_3$
15 **return** F

Algorithm 2: Update index - match

Input: A new trip request r_n and the matched vehicle c_i
1 **if** c_i *empty* **then**
2 $T_{ev}.remove(c_i)$
3 **else**
4 **for** *a segment in the trip schedule of* c_i **do**
5 remove the ellipse of the *segment* from T_{seg}
6 T_{end}.remove(ending stop of c_i)
7 add $r_n.s$ and $r_n.e$ to the trip schedule of c_i
8 **for** *a segment in the trip schedule of* c_i **do**
9 compute the detour ellipse of the *segment*
10 insert the ellipse of the *segment* into T_{seg}
11 T_{end}.insert(the end stop of c_i)

Algorithm 3: Update index - move

Input: A moving vehicle c_i
1 $P \leftarrow$ obsolete segments of c_i
2 **for** $p \in P$ **do**
3 $T_{seg}.remove(p)$
4 **if** c_i *reaches the ending stop* **then**
5 $T_{end}.remove$(ending stop of c_i)
6 $T_{ev}.insert(c_i)$

4.1 Algorithm Complexity

We measure the complexity of our algorithm by two parameters $|S|$ and C that are key to our algorithm: $|S|$ is the maximum number of

stops of the vehicle schedules (a small constant constrained by the vehicle capacity) and $|C|$ is the number of vehicles. We note that instead of using $|S|$, we may also use the number of requests $|R|$ because there is a linear relationship: $|S||C| \propto |R|$. The state-of-the-art pruning algorithms [19, 25] lack complexity analysis.

Pruning. It takes $O(1)$ time to compute the waiting circle and the detour ellipse of a new request. There are at most $|S||C|$ MBRs in T_{seg} and $|C|$ entries in T_{end}. The point query on T_{seg} returns at most $|S||C|$ results and hence the complexity is $O(\sqrt{|S||C|} + |S||C|)$ [20]. At most $|C|$ results will be returned from the range query on T_{end} and the complexity is $O(\sqrt{|C|} + |C|)$. The time complexity of the queries on R-trees is thus $O(\sqrt{|S||C|} + |S||C|)$. Checking the time and capacity constraints takes $O(|S||C| + |C|)$ time.

It takes $O(|S||C| + |C|)$ time to retrieve the corresponding vehicles and at most $|C|$ vehicles will be returned in each set after sorting ($O(|S||C| \log(|S||C|))$ time). The set intersection hence takes O($|C|$) time [12]. The overall time complexity of GeoPrune is thus $O(\sqrt{|S||C|} + |S||C| \log(|S||C|))$.

Update. When a new trip request is assigned to a vehicle c_i, it takes $O(\log |C|)$ time to delete c_i from T_{ev} if c_i was empty, $O(|S| \log (|S||C|))$ time to remove invalid segments from T_{seg}, and $O(\log |C|)$ time to remove the obsolete record in T_{end} [20]. For the new schedule of c_i, there are at most $|S|$ new segments. It thus takes $O(|S|)$ time to compute the new detour ellipses for the new segments and $O(|S| \log^2(|S||C|))$ time to insert the ellipses to T_{seg} [20]. The overall update time for a new request is $O(|S| \log^2(|S||C|))$.

When a vehicle moves, the number of obsolete scheduled stops is at most $|S|$. Therefore, the time to remove obsolete vehicle ellipses from T_{seg} is $O(|S| \log(|S||C|))$. At most $|C|$ vehicles change their status while moving, hence the time to update T_{end} and T_{ev} is at most $O(|C| \log^2 |C|)$. Therefore, the overall update time for moving all vehicles in a time slot is $O(|S| \log(|S||C|) + |C| \log^2 |C|)$.

5 EXPERIMENTS

In this section, we study the empirical performance of GeoPrune and compare it against the state-of-the-art. All algorithms are implemented in C++ r on a 64-bit virtual node with a 1.8 GHz CPU and 128 GB memory from an academic computing cloud (Nectar [3]) running on OpenStack. The travel distance between points is computed by a shortest path algorithm on road networks [7].

5.1 Experimental Setup

Dataset. We perform the experiments on real-world road network datasets extracted from OpenStreetMap [4], *New York City* (NYC) and *Chengdu* (CD). We transform the coordinates to Universal Transverse Mercator (UTM) coordinates to support pruning based on Euclidean distance. We use real-world taxi requests on the two road networks [1, 2] and remove unrealistic ones, i.e., duration time less than 10 seconds or longer than 6 hours. There are 448,128 taxi requests (April 09, 2016) for NYC and 259,423 (November 18, 2016) taxi requests for Chengdu. Every request consists of a source, a destination, and an issue time. We map the locations to their nearest road network vertices. Similar to previous studies [10, 17], we assume the number of passengers to be one per request.

Implementation. We run simulations following the settings of previous studies [17, 27]. The vehicle initial positions are randomly selected from the road network vertices. Non-empty vehicles move on the road network following their trip schedules (shortest paths) while empty vehicles stay at their last drop-off location until they are committed to new requests. Similar to previous studies [10, 17], we use a constant travel speed for all edges (48km/h). For the selection step, we apply the state-of-the-art insertion algorithm [27] to minimize the total travel distance for all methods. If no satisfying vehicle is found for a new trip request, the trip request is ignored. We use in-memory R-trees for indexing.

Table 3: Datasets

Name	# vertices	# edges	# requests
NYC	166,296	405,460	448,128
CD	254,423	467,773	259,343

Table 4: Experiment parameters

Parameters	Values	Default
Number of vehicles	2^{10} to 2^{17}	2^{13}
Waiting time (min)	2, 4, 6, 8, 10	4
Detour ratio	0.2, 0.4, 0.6, 0.8	0.2
Number of requests	20k to 100k	60k
Frequency of requests (# requests/second)	1 to 10	refer to table 3
Transforming speed (km/h)	20 to 140	48

By default (Table 4), we simulate ride-sharing on 2^{13} vehicles with a capacity of 4 and 60,000 trip requests, and the maximum waiting time and the detour ratio are 4 min and 0.2.

Baselines. We compare with the following state-of-the-art pruning algorithms using their originally reported parameter values.

(1) **GreedyGrids [27].** This algorithm retrieves all vehicles that are currently in the nearby grid cells.

(2) **Tshare [19].** This is the single-side search algorithm of Tshare. (their dual-side search algorithm terminates when a feasible vehicle is found while we aim to find all feasible vehicles). The grid cell lengths of both GreedyGrids and Tshare are set to 1 km [27].

(3) **Xhare [25].** This algorithm only checks non-empty vehicles. For a fair comparison, we prune empty vehicles in Xhare using the same algorithm applied in our method (see Section 3.4). We optimize the update process by precomputing the pair-wise distance between clusters. The landmark size is set to 16,000 for NYC and 23,000 for Chengdu, and the grid length is set to 10 m. The maximum distance between landmarks in a cluster is set to 1 km.

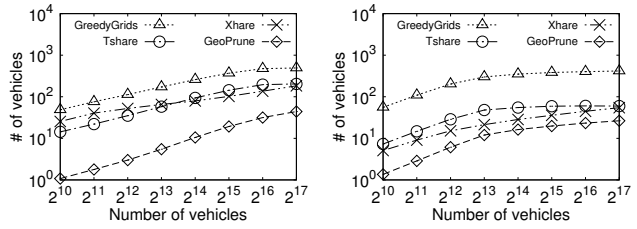
Metrics. We measure and report the following metrics:

(1) *Number of remaining vehicles* – the number of remaining candidate vehicles after the pruning. Note that GeoPrune prunes empty vehicles and non-empty vehicles separately with different criteria, and such a scheme is applied on Xhare to make it applicable. GreedyGrids and Tshare, however, process the two types of vehicles together and return both types after pruning. For consistency, we only compare the number of remaining non-empty vehicles.

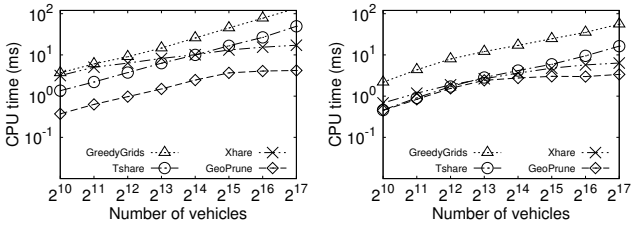
(2) *Match time* – the total running time of the matching process, including both pruning and selection time.

(3) *Overall update time* – the overall match update and move update time.

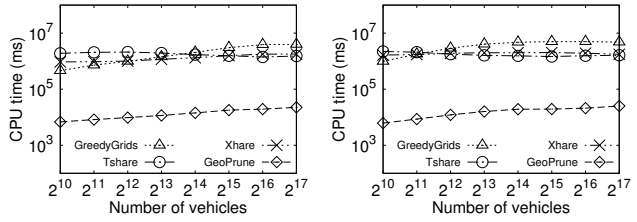
(4) *Memory consumption* – the memory cost of the data structures of an algorithm.



(a) # remaining vehicles (NYC). (b) # remaining vehicles (CD).

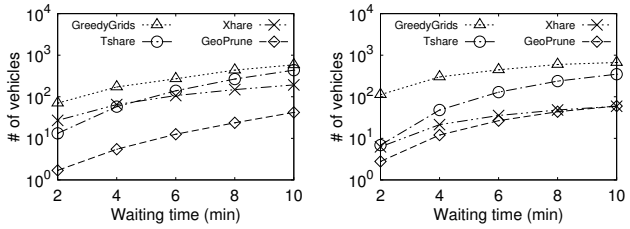


(c) Overall match time (NYC). (d) Overall match time (CD).

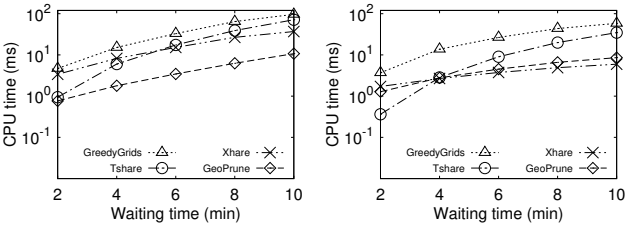


(e) Overall update time (NYC). (f) Overall update time (CD).

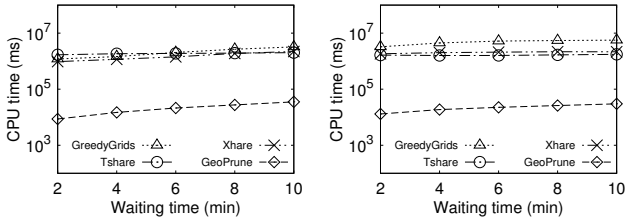
Figure 7: Effect of the number of vehicles.



(a) # remaining vehicles (NYC). (b) # remaining vehicles (CD).



(c) Overall match time (NYC). (d) Overall match time (CD).



(e) Overall update time (NYC). (f) Overall update time (CD).

Figure 8: Effect of the waiting time.

As we use the state-of-the-art selection algorithm [27], we obtain the same matching quality as GreedyGrids [27], including total increased travel distance (our optimization goal) and matching ratio. Tshare and Xshare are approximate algorithms and the false negatives may randomly impact the matching quality. We omit detailed matching quality results as we focus on pruning. We also omit the results on varying the capacity due to the space limit and the stable behavior of all algorithms (as observed in [27, 28]). GeoPrune consistently outperforms others in all capacity settings.

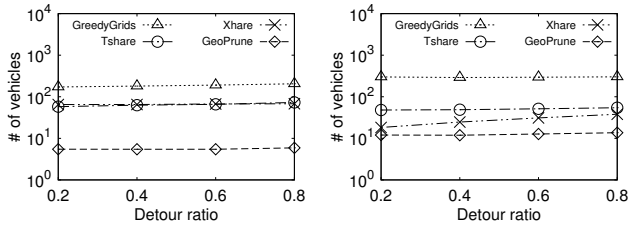
5.2 Experimental Results

5.2.1 Effect of the Number of Vehicles. Figure 7 shows the results on varying the number of vehicles. GeoPrune substantially reduces the number of remaining candidate vehicles. When there are 2^{13} vehicles, the average number of candidates of GeoPrune is only 5 on the NYC dataset, while the other algorithms return 57 ∼ 172 candidates per request. GreedyGrids returns the largest set of candidates as it retrieves all vehicles in nearby grid cells, among which only a few are feasible. Tshare and Xhare find fewer candidates than GreedyGrids but may result in false negatives due to approximation. Tshare and Xhare perform better on Chengdu than on NYC. The reason might be that requests of NYC have a higher frequency than those of Chengdu, while Tshare and Xhare are more sensitive to the frequency of trip requests (consistent with our results in Figure 11).

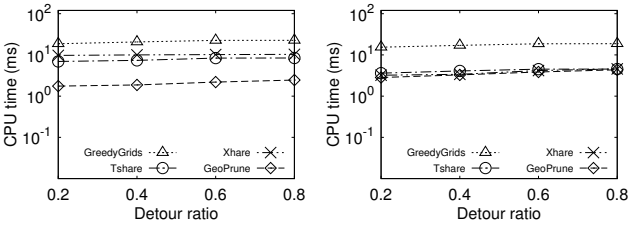
The number of remaining vehicles largely affects the running time of the selection stage and the overall match time. As shown in Figure 7c and Figure 7d, GeoPrune reduces the overall match time by 71% to 90% on the NYC dataset and up to 80% on the Chengdu dataset. Consistent with experiments shown in the previous studies [22, 27, 28], all algorithms exhibit longer pruning time with more vehicles as the number of vehicle candidates increases. The match time of Tshare and Xhare is comparable with GeoPrune on the Chengdu dataset when the number of vehicles is small but continuously increases with more vehicles, showing that GeoPrune scales better with the increase in the number of vehicles.

As for the update cost, GeoPrune is two to three orders of magnitude faster since it only relies on circles and ellipses for pruning while others need real-time maintenance of indices on the networks.

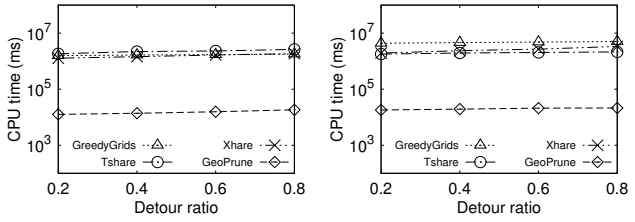
5.2.2 Effect of the Maximum Waiting Time. Figure 8 shows the experimental results when varying the maximum waiting time. All algorithms exhibit longer times with the increasing waiting time because of larger shareability between requests and more returned vehicle candidates. GeoPrune again shows the best pruning performance in almost all cases. Tshare requires less match time than GeoPrune when the waiting time is 2 min on the Chengdu dataset. However, longer waiting time requires Tshare to check more grid cells and continuously increase their match time, which becomes five times slower than GeoPrune when the waiting time is 10 min. Xhare finds fewer vehicles and requires less match time than GeoPrune when the waiting time is longer than 6 min on the Chengdu dataset. This is because Xhare assumes vehicles travel on predefined routes, and new requests can only be served on the way of these routes. A long waiting time brings more feasible vehicles with append-append case and Xhare may miss these vehicles.



(a) # remaining vehicles (NYC). (b) # remaining vehicles (CD).

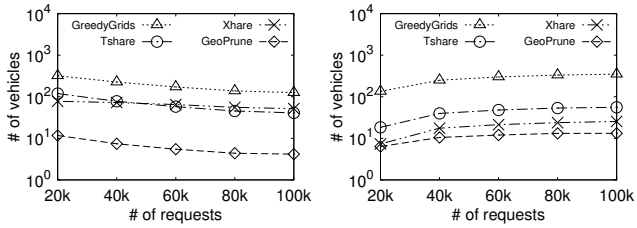


(c) Overall match time (NYC). (d) Overall match time (CD).

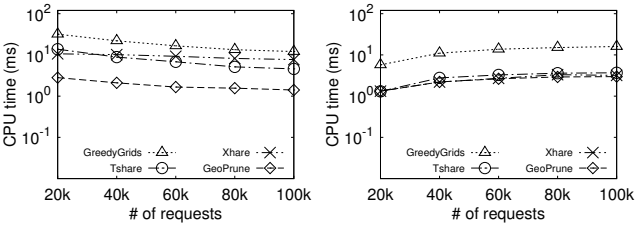


(e) Overall update time (NYC). (f) Overall update time (CD).

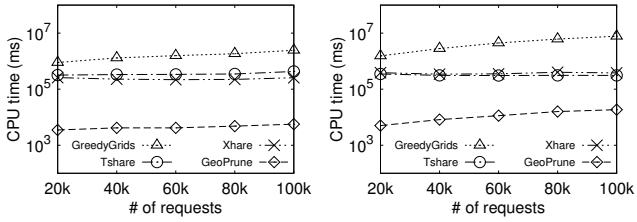
Figure 9: Effect of the detour ratio.



(a) # remaining vehicles (NYC). (b) # remaining vehicles (CD).



(c) Overall match time (NYC). (d) Overall match time (CD).



(e) Overall update time (NYC). (f) Overall update time (CD).

Figure 10: Effect of the number of requests.

Figure 8e and Figure 8f show the update cost, which increases for all algorithms with the larger waiting time as the vehicle schedules become longer and more requests can be shared. Still, GeoPrune is two to three orders of magnitude faster on update than others.

5.2.3 Effect of the Detour Ratio. Figure 9 shows the sensitivity over the detour ratio. Again, GeoPrune prunes more infeasible vehicles and its match time is three to ten times faster than others on the NYC dataset and comparable with Tshare and Xhare on the Chengdu dataset. The number of remaining vehicles of all algorithms keeps almost stable due to the limited shareability. The update cost of all algorithms remains stable (and three orders of magnitude smaller for GeoPrune) as the length of vehicle schedules is barely affected.

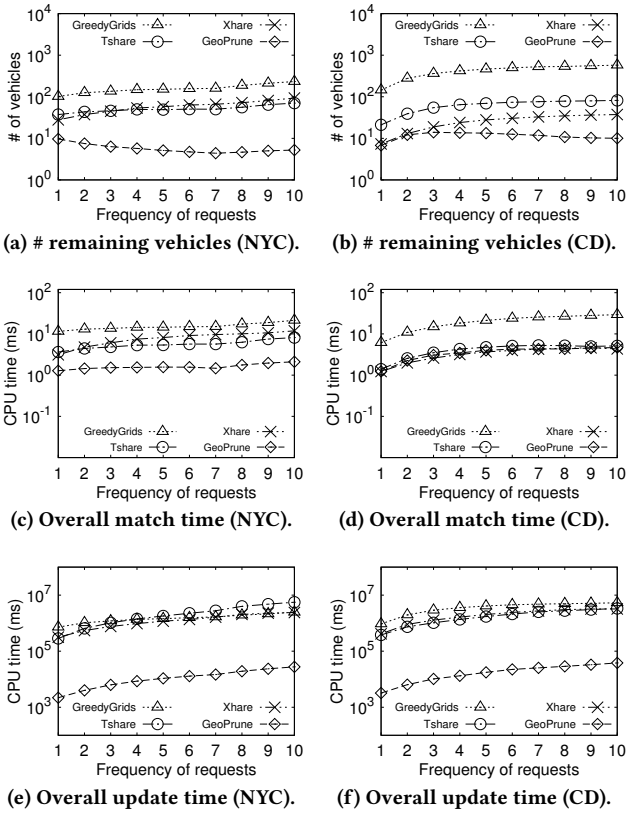
5.2.4 Effect of the Number of Trip Requests. Figure 10 shows the experiments when the number of requests varies. Interestingly, algorithms show different behavior on the two datasets. When the number of requests changes from 20 k to 100 k, the candidates returned by GeoPrune for each request decreases from 11 to 4 on the NYC dataset but increases from 6 to 13 on the Chengdu dataset, meaning that the shareability between requests decreases on the NYC dataset while increases on the Chengdu dataset (may be caused by the different geographical distribution of requests and vehicles). The trend of the match time is consistent with that of the number of remaining vehicles, which again confirms that the match time is largely affected by the pruning effectiveness.

More trip requests correspond to longer simulation time and increase the total update cost (with GeoPrune still being two to three orders of magnitude cheaper in terms of update cost).

5.2.5 Effect of the Trip Request Frequency. Figure 11 shows the scalability of algorithms with the frequency of trip requests varying from 1 to 10 requests per second over 3 hours. Note that the frequencies of the original NYC and Chengdu datasets are 5.19 and 3 requests per second respectively. To generate trip requests less frequent than the original datasets, we uniformly sample trip requests from the original datasets. As for more frequent trip requests, we extract a certain number of trip requests according to the frequency, e.g., 10,800×7=75,600 trip requests when the frequency is 7. We then uniformly distribute the request issue time over 3 hours.

All algorithms return more vehicle candidates with higher frequency while GeoPrune keeps almost stable, showing that GeoPrune provides tighter pruning and is more scalable to dynamic scenarios. The match time of GeoPrune consistently outperforms others on NYC dataset and is comparable with Tshare and Xhare on Chengdu dataset. The update cost of all algorithms grows with higher frequency due to more frequent updates while GeoPrune again outperforms others by two to three orders of magnitude.

5.2.6 Effect of the Transforming Speed: All algorithms need a speed value to transform the time constraint to distance constraint so that pruning based on geographical locations can be applied. Figure 12 shows the effect of the transforming speed. A higher speed enlarges the search space and thus all algorithms show longer match time. However, GeoPrune consistently performs efficient pruning with all speed values. Figure 12b shows the total number of false negatives wrongly pruned for 60,000 requests. Same as GreedyGrids,

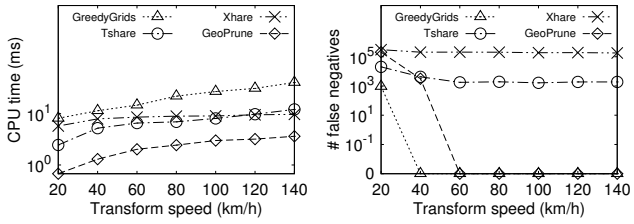


(a) # remaining vehicles (NYC). (b) # remaining vehicles (CD).

(c) Overall match time (NYC). (d) Overall match time (CD).

(e) Overall update time (NYC). (f) Overall update time (CD).

Figure 11: Effect of the frequency of requests.



(a) Overall match time. (b) # of false negatives.

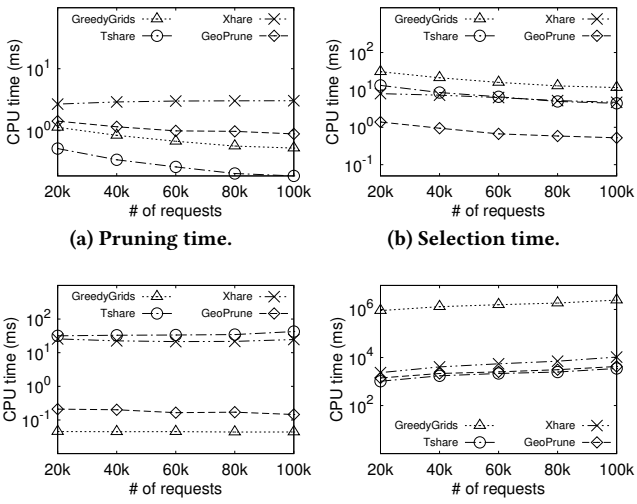
Figure 12: Effect of the transforming speed (NYC).

Table 5: Memory consumption (MB) (# vehicles = 2^{13}).

	GreedyGrids	Tshare	Xhare	GeoPrune
NYC	0.38	100.34	1546.40	6.56
Chengdu	1.67	9965.37	21282.46	6.43

GeoPrune ensures no false negatives when the speed is greater than vehicle speed (48km/h), whereas Xhare and Tshare still have false negatives even with high transforming speed. This verifies the robustness of GeoPrune on real-time traffic conditions, where the transforming speed is set as the maximum speed so that the pruning is still correct and the processing time only increases slightly.

5.2.7 Memory Consumption. Table 5 illustrates the maximum memory usage of the algorithms under the default setting. All state-of-the-arts consume more memory on the Chengdu dataset as it has a large road network, while GeoPrune keeps stable. For example, the grid size of Tshare in NYC is 46×46 but increases to 174×174 in Chengdu. GeoPrune, however, only maintains several R-trees and thus is less affected by the road network size. GreedyGrids has the smallest memory footprint as it only records a list of in-cell vehicles for each grid cell, which is consistent with the observation in [27]. Tshare and Xhare consume much more memory than GeoPrune due to the large road network index.



(a) Pruning time. (b) Selection time.

(c) Match update time. (d) Move update time.

Figure 13: The cost breakdown of algorithm steps.

5.2.8 Cost Breakdown of Algorithm Steps. Figure 13 compares the cost of different phases in the match process and update process when varying the number of requests on the NYC dataset. Figure 13a shows the cost of the pruning algorithms while Figure 13b shows the selection cost based on their pruning results. GeoPrune requires a slightly longer time for pruning than Tshare and Xhare but can reduce the selection time by more than 88% due to the fewer remaining vehicles. The selection time of algorithms (Figure 13b) is consistent with the number of remaining vehicles (Figure 10a), which again demonstrates that the selection step is largely affected by the number of remaining vehicles.

Figure 13c shows the update cost when a request is newly assigned. GeoPrune is slightly slower than GreedyGrids to update the R-trees. Xhare and Tshare, however, need much more time than GeoPrune to update the reachable areas of the matched vehicle while GeoPrune quickly bounds the areas by ellipses.

Figure 13d compares the update cost when vehicles are moving. GreedyGrids is two orders of magnitude slower than the other three algorithms because it needs to track the located grid cells of continuously moving vehicles.

6 RELATED WORK

Dynamic ride-sharing matching has been studied with different optimization goals. A common optimization goal is to minimize the total travel cost of vehicles [8, 17, 19, 25, 27]. A few other studies aim to provide a better service experience to passengers [11, 14, 28, 29]. Some studies maximize the overall profit of the ride-sharing platform [9, 30, 31] and the number of served requests [18]. A common need in these problems is to efficiently filter out infeasible

vehicles that violate the constraints, such that the optimal vehicles from the remaining ones can be computed with lower costs.

We focus on the studies that aim to minimize the total travel cost as we use it to examine our algorithm. Huang et al. [17] maintain a *kinetic tree* for each vehicle to record all possible routes instead of a single optimal route. GeoPrune can be easily integrated into their setting by computing the detour ellipses of all possible routes. Alonso et al. [8] assign requests to vehicles in batches. They first compute the shareability between requests and vehicles and then construct a graph to connect shareable requests and vehicles. Their shareability computation requires an exhaustive check, which can be streamlined by GeoPrune . The state-of-the-art selection algorithm [27] for minimizing the total vehicle travel time first filters infeasible vehicles by checking whether inserting the new request to the vehicle schedules is valid based on the Euclidean distance. It then ranks all remaining vehicles using the increased distance computed based on the Euclidean distance and sequentially checks these remaining vehicles using road network distances. Although this algorithm has a pruning step, GeoPrune can further improve it by reducing the number of vehicles for individual checking.

Next, we discuss existing algorithms for pruning infeasible vehicles – Tshare [19] and Xhare [25]. Tshare builds an index over the road network by partitioning the space into equal-sized grid cells. The distance between two objects (e.g., a request and a vehicle) is estimated using the centers of their corresponding cells. Such an estimation may miss feasible vehicles. The geometric objects applied in GeoPrune, in comparison, bound the reachable areas and ensure all feasible vehicles to be returned. Moreover, Tshare stores pairwise distances between all grid cells and does not scale to large networks due to the high memory cost. Besides, Tshare maintains the pass-through grid cells of vehicles in real-time, which is costly for highly dynamic scenarios and inflexible to the change of traffic conditions. In comparison, GeoPrune can quickly bound the reachable areas using ellipses and index these areas using R-trees, which saves computation and update costs. Xhare partitions the road network into three levels: grid cells, landmarks, and clusters. The reachable areas of vehicles are estimated using the distance between clusters. Similar to Tshare, Xhare is an approximate method. The index of Xhare may have a large memory footprint for large networks and high update cost for dynamic scenarios.

7 CONCLUSIONS

We studied the dynamic ride-sharing matching problem and proposed an efficient algorithm named GeoPrune to prune infeasible vehicles to serve trip requests. Our algorithm applies circles and ellipses to bound the areas that vehicles can visit without violating the service constraints of passengers. The circles and ellipses are simple to compute and further indexed using efficient data structures, which make GeoPrune highly efficient and scalable. Our experiments on real-world datasets confirm the advantages of GeoPrune in pruning effectiveness and matching efficiency. In the future, it is worth exploring the advantages of GeoPrune in other ride-sharing settings or to other spatial crowd-sourcing problems.

ACKNOWLEDGMENTS

This research was partially supported under Australian Research Council's Discovery Projects funding scheme (project number DP180103332).

REFERENCES

[1] 2016. GAIA Open Dataset. https://outreach.didichuxing.com/appEn-vue/DataList
[2] 2017. TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[3] 2020. Nectar Cloud. http://nectar.org.au/research-cloud/
[4] 2020. OpenStreetMap. https://www.openstreetmap.org/
[5] 2020. Uber Revenue and Usage Statistics. http://www.businessofapps.com/data/uber-statistics/
[6] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. 2016. K-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *PVLDB* 9, 6 (2016), 492–503.
[7] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. 2014. Fast shortest-path distance queries on road networks by pruned highway labeling. In *ALENEX*. 147–154.
[8] Javier Alonso-Mora, Samitha Samaranayake, Alex Wallar, Emilio Frazzoli, and Daniela Rus. 2017. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *PNAS* 114, 3 (2017), 462–467.
[9] Mohammad Asghari, Dingxiong Deng, Cyrus Shahabi, Ugur Demiryurek, and Yaguang Li. 2016. Price-aware real-time ride-sharing at scale: an auction-based approach. In *SIGSPATIAL*. 3.
[10] Lu Chen, Qilu Zhong, Xiaokui Xiao, Yunjun Gao, Pengfei Jin, and Christian S Jensen. 2018. Price-and-time-aware dynamic ridesharing. In *ICDE*. 1061–1072.
[11] Peng Cheng, Hao Xin, and Lei Chen. 2017. Utility-aware ridesharing on road networks. In *SIGMOD*. 1197–1210.
[12] Bolin Ding and Arnd Christian König. 2011. Fast set intersection in memory. *PVLDB* 4, 4 (2011), 255–266.
[13] Zhiming Ding and Ralf Hartmut Guting. 2004. Managing moving objects on dynamic transportation networks. In *SSDBM, 2004.* IEEE, 287–296.
[14] Xiaoyi Duan, Cheqing Jin, Xiaoling Wang, Aoying Zhou, and Kun Yue. 2016. Real-time personalized taxi-sharing. In *DASFAA*. 451–465.
[15] Preeti Goel, Lars Kulik, and Kotagiri Ramamohanarao. 2016. Privacy-aware dynamic ride sharing. *ACM Transactions on Spatial Algorithms and Systems (TSAS)* 2, 1 (2016), 4.
[16] Antonin Guttman. 1984. R-trees: A dynamic index structurefor spatial searching. In *SIGMOD*. 47–57.
[17] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. 2014. Large scale real-time ridesharing with service guarantee on road networks. *PVLDB* 7, 14 (2014), 2017–2028.
[18] Zhidan Liu, Zengyang Gong, Jiangzhou Li, and Kaishun Wu. 2020. Mobility-Aware Dynamic Taxi Ridesharing. In *ICDE*.
[19] Shuo Ma, Yu Zheng, and Ouri Wolfson. 2013. T-share: A large-scale dynamic taxi ridesharing service. In *ICDE*. 410–421.
[20] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N Papadopoulos, and Yannis Theodoridis. 2010. *R-trees: Theory and Applications*. Springer Science & Business Media.
[21] Neda Masoud and R Jayakrishnan. 2017. A real-time algorithm to solve the peer-to-peer ride-matching problem in a flexible ridesharing system. *Transportation Research Part B: Methodological* 106 (2017), 218–236.
[22] James J Pan, Guoliang Li, and Juntao Hu. 2019. Ridesharing: simulator, benchmark, and evaluation. *PVLDB* 12, 10 (2019), 1085–1098.
[23] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. 2003. Query processing in spatial network databases. In *PVLDB*. 802–813.
[24] Kevin Shaw, Elias Ioup, John Sample, Mahdi Abdelguerfi, and Olivier Tabone. 2007. Efficient approximation of spatial network queries using the m-tree with road network embedding. In *SSDBM*. 11–11.
[25] Raja Subramaniam Thangaraj, Koyel Mukherjee, Gurulingesh Raravi, Asmita Metrewar, Narendra Annamaneni, and Koushik Chattopadhyay. 2017. Xhare-a-Ride: A search optimized dynamic ride sharing system with approximation guarantee. In *ICDE*. 1117–1128.
[26] Yongxin Tong, Libin Wang, Zimu Zhou, Lei Chen, Bowen Du, and Jieping Ye. 2018. Dynamic pricing in spatial crowdsourcing: A matching-based approach. In *SIGMOD*. 773–788.
[27] Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, Jieping Ye, and Ke Xu. 2018. A unified approach to route planning for shared mobility. *PVLDB* 11, 11 (2018), 1633–1646.
[28] Yi Xu, Yongxin Tong, Yexuan Shi, Qian Tao, Ke Xu, and Wei Li. 2019. An Efficient Insertion Operator in Dynamic Ridesharing Services. In *ICDE*. 1022–1033.
[29] Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2019. Last-mile delivery made practical: an efficient route planning framework with theoretical guarantees. *Proceedings of the VLDB Endowment* 13, 3 (2019), 320–333.
[30] Libin Zheng, Lei Chen, and Jieping Ye. 2018. Order dispatch in price-aware ridesharing. *PVLDB* 11, 8 (2018), 853–865.
[31] Libin Zheng, Peng Cheng, and Lei Chen. 2019. Auction-based order dispatch and pricing in ridesharing. In *ICDE*. 1034–1045.
[32] Ming Zhu, Xiao-Yang Liu, and Xiaodong Wang. 2018. An online ride-sharing path-planning strategy for public vehicle systems. *TITS* 20, 2 (2018), 616–627.