Geoinformatica (2020) 24:881–913
https://doi.org/10.1007/s10707-020-00411-0

SHAREK*: A Scalable Matching Method for Dynamic Ride Sharing

Bin Cao[1] · Chenyu Hou[1] · Liwei Zhao[1] · Louai Alarabi[2] · Jing Fan[1] · Mohamed F. Mokbel[3] · Anas Basalamah[4]

Received: 27 February 2018 / Revised: 31 March 2020 / Accepted: 24 April 2020 /
Published online: 2 June 2020

Abstract

Due to its significant economic and environmental impact, sharing the ride among a number of drivers (i.e., car pooling) has recently gained significant interest from industry and academia. Hence, a number of ride sharing services have appeared along with various algorithms on how to match a rider request to a driver who can provide the ride sharing service. However, existing techniques have several limitations that affect the quality of the ride sharing service, and hence hinder its wide applicability. This paper proposes SHAREK*; a scalable and efficient ride sharing service that overcomes the limitations of existing approaches. SHAREK* allows riders requesting the ride sharing service to indicate

✉ Jing Fan
fanjing@zjut.edu.cn

Bin Cao
bincao@zjut.edu.cn

Chenyu Hou
houcy@zjut.edu.cn

Liwei Zhao
otoko_zlw@foxmail.com

Louai Alarabi
lmarabi@uqu.edu.sa

Mohamed F. Mokbel
mokbel@cs.umn.edu

Anas Basalamah
ambasalamah@uqu.edu.sa

1 Zhejiang University of Technology, HangZhou, China

2 Department of Computer Science, Umm Al-Qura University, Mecca, Kingdom of Saudi Arabia

3 Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA

4 Computer Engineering Department and KACST GIS Technology Innovation Center, Umm Al-Qura University, Makkah, Saudi Arabia

the maximum price they are willing to pay for the service, the maximum waiting time before being picked up and the maximum arrival time for arriving the destination. In the mean time, SHAREK* computes the price of the service based on the distance of the rider trip and the detour that the driver will make to offer the service. Then, SHAREK* returns a set of drivers that can make it to the rider within its price and temporal constraints. Since there could be many of such drivers, SHAREK* internally prunes those drivers that are dominated by others, i.e., they provide higher price and higher waiting time (or arrival time) than other drivers. To realize its efficiency and scalability, SHAREK* employs a set of early pruning techniques that minimize the need for any actual shortest path computations.

Keywords Ride sharing · Dynamic matching · Skyline

1 Introduction

Dynamic ride sharing can be viewed as a form of car pooling system that arranges ad-hoc shared rides with sufficient convenience and flexibility [11]. Since dynamic ride sharing can be enabled by smart mobile phones, GPS and wireless networks, it is viewed as an environmentally and socially sustainable way to solve the world-widely major transportation problems, such as finite oil supplies, high gas prices, and jam-packed traffic. With the increasing number of the vehicles, it is widely believed that dynamic ride sharing will gain more popularity in the coming years.

The significance of the dynamic ride sharing attracts the interests from both industry and academia [1, 15, 22]. As a result, a number of dynamic ride sharing systems are available nowadays, e.g., Flinc [12], Lyft [20], Noah [28]. However, the way that current ride sharing systems match drivers to requesting riders suffer from one or more of the following drawbacks: (1) The matching models are quite simple and limited. For example, some systems just select the nearest k drivers to the rider for picking up. In these systems, close by drivers may have way far destinations than the rider, and hence they are not suitable for ride sharing. Meanwhile, there exists systems that require the rider route is part of the driver route, where some important drivers that may have only a small detour to suit the rider request will not be reported. (2) The cost of the ride sharing service is not considered during the matching and it is left to be negotiated between the riders and drivers in a personal way. This is very problematic as it may be the case that the most convenient drivers to pick up the rider have higher costs that is beyond what the rider would like to pay. (3) The speedup technique mainly depends on precomputing the driver routes based on their historical trajectories. This indeed works to some extent, however, it would fail when the drivers' trajectories are missing. Besides, storing those trajectories needs massive storage which adds new cost to the operators of the ride sharing systems.

In this paper, we present SHAREK*, a new scalable ride sharing system that avoids the drawbacks of all previous approaches. SHAREK* matches a rider, requesting a ride sharing service, to a set of drivers who can provide the requested ride sharing service, while taking into account: (a) the cost of the ride sharing service, and (b) the convenience of the service for both the driver and the rider. SHAREK* allows drivers willing to offer a ride sharing service to register themselves indicating their current source and destination locations, e.g., a driver is going back from work to home. For example, Fig. 1 shows a typical ride sharing scenario where black points represent the drivers and white points represent the rider. A dotted vertical line separates these points into two parts. The points in the left part are origins of the drivers and the rider while the right part contains their destinations. Moreover,

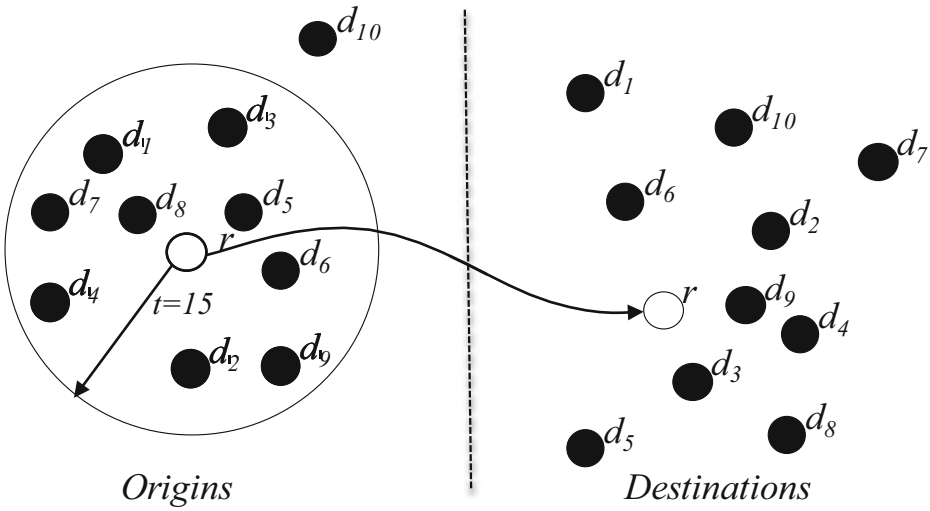


Fig. 1 A SHAREK* example

SHAREK* allows riders requesting the ride sharing service to indicate their destinations as well as to express two main constraints: (1) *Cost constraint*. The maximum price the rider is willing to pay for the ride sharing service, and (2) *Temporal constraint*. The maximum waiting time that the rider can wait before being picked up by the driver. Then, SHAREK* employs a cost model that estimates the cost of the ride sharing service for each driver. The cost model is basically computed based on the distance of the rider route in addition to the additional distance overhead that the driver will encounter to detour from his original route to accommodate the rider request. Based on the cost model, SHAREK* can pinpoint those drivers that can make it to the rider within its cost and temporal constraints. Since there could be many of such drivers, SHAREK* internally prunes those drivers that are dominated by others. For example, if two drivers d_i and d_j satisfy both the cost and temporal constraints of the rider, yet d_i would result in less cost and less waiting time than d_j, we say that d_i dominates d_j, i.e., d_j is *not* on the skyline set of drivers who satisfy the rider constraints. Hence, we prune d_j and do *not* report it as a candidate driver. SHAREK* only reports those drivers that are *not* dominated by others, i.e., the list of skyline drivers according to cost and temporal constraints.

One trivial way to realize SHAREK* vision is to calculate the actual cost and waiting time that each possible driver d would offer to the rider r. Then, run a skyline algorithm over all of the drivers. Such trivial way is prohibitively expensive as it encounters a large number of road network shortest path computations. SHAREK* achieves its scalability through minimizing the need to rely on the expensive shortest path operation. In fact, SHAREK* can efficiently and accurately satisfy the ride sharing request with only few shortest path computations by applying filter-and-refine paradigm [33]. To do so, SHAREK* employs three consecutive phases. In the first phase (*Euclidian Temporal Pruning*), we take advantage of the rider temporal constraint to prune a set of drivers without computing any road network shortest path operation. In the second phase (*Euclidean Cost Punning*), we employ a conservative Euclidean computations to prune a set of drivers based on the rider cost constraint, without computing any road network shortest path. In the third phase (*Semi-Euclidean Skyline-aware pruning*), we start to compute actual road network shortest paths

in a very conservative way. Meanwhile, different from previous works [28, 32], we inject the skyline computations inside the pruning techniques, which helps in pruning even more drivers without any shortest path computations. So, instead of considering the skyline computation as an overhead, we actually consider it as a blessing, where we take advantage of it to even prune more drivers without further computations.

Extensive experimental evaluation show the scalability and efficiency of SHAREK*. It only takes tens of milliseconds to satisfy the rider request, even if there are 100,000 drivers around. Experimental analysis also shows the pruning power of each phase in SHAREK*, and show that we can get the set of candidate drivers satisfying all rider constraints with very few shortest path computations. In general, the contributions in this paper can be summarized as follows:

1. We define the ride sharing problem in a way that accommodates the rider convenience by expressing temporal and cost constraints along with defining a price cost model for each ride sharing service.
2. We introduce an efficient and scalable algorithm for the ride sharing service that: (a) takes into account the rider temporal and cost constraints and (b) avoids reporting unnecessary large number of candidate drivers that may satisfy the rider constraints by reporting only the skyline set of those drivers in terms of price and waiting time.
3. SHAREK* distinguishes itself from SHAREK [6] in the following points: (1) SHAREK* supports more temporal constraints from the rider, i.e., the maximum waiting time and the maximum arrival time. (2) SHAREK* provides an alternative implementation for the *Semi-Euclidean Skyline-aware pruning* phase, which is able to show a better performance when the cost constraint is below a certain value.
4. We provide experimental evidence of the scalability and efficiency of our proposed algorithm.

The rest of this paper is organized as follows. Section 2 sets the stage for various concepts used in SHAREK*. Section 3 discusses SHAREK* query processing. Section 4 gives an alternative implementation for SHAREK*. Experimental evaluation is presented in Section 5. Section 6 highlights related work. Finally, Section 7 concludes the paper.

2 Preliminaries

This section presents a set of preliminaries that are important to set the stage for understanding SHAREK* and its vision. In particular, we discuss what we mean in SHAREK* by drivers and riders, the concept of skyline drivers, the price cost model, the problem definition, and the underlying data structure.

2.1 Drivers and Riders

Users of SHAREK* are either *drivers* or *riders*, as below:

Drivers The set of driver D represents the ride sharing service providers. Drivers are ordinary people who are just commuting in their daily life. At one point, they may indicate their willingness to offer a ride sharing service within their route. To do so, they call SHAREK* service to register themselves indicating their origin $orig$ and destination points $dest$. After registration, SHAREK* is allowed to track their locations for supporting the ride sharing service. Once the driver reaches to his destination, the driver is unregistered

from SHAREK*. Every time the driver is willing to share his route, he has to register again with SHAREK*. This is done through a simple check-in mobile app. Drivers are to be paid for their ride sharing service based on the distance of the ride service they will provide and the detour that they will need to make from their original route.

Riders The set of riders R represents people requesting a ride sharing service. To request such service, a rider r would call SHAREK* service, through the dedicated mobile app, and provide four pieces of information: (1) current location $orig$, which can be obtained directly from the cell phone, (2) the requested destination $dest$, (3) the maximum waiting time ($max\_WaitTime$) that r can afford before being picked up, (4) the maximum arrival time ($max\_ArrivalTime$) that r is able to spare for arriving his destination, and (5) the maximum price (max_Price) that r is willing to pay for ride sharing service. Within few milliseconds, the rider receives a set of drivers from SHAREK* that can offer the requested ride sharing service within the arrival time and price constraints.

2.2 Skyline Drivers

For a certain ride sharing request from a rider r, there could be more than one driver capable of satisfying r requested within its waiting time, arrival time and price constraints. It is challenging then which of these capable drivers to return to the rider. Should we decide to return only the driver with least waiting and arrival time, we may end up in returning an expensive driver, though it is still within the rider cost constraints. Similarly, the driver with cheapest price may end up on the highest waiting and arrival time. In the mean time, returning all possible candidate drivers satisfying the rider constraints may not be practical and result in redundant information.

Hence, SHAREK* opts to use the logic of the maximal vector set problem [18] (also known as the *skyline* query in database context [5]) to return only the set of the three dimensioanl *skyline* drivers in terms of waiting time, arrival time and price. A driver d_i belongs to the set of *skyline* drivers if there is no other driver d_j that has less waiting time, arrival time as well as price than that of d_i simultaneously. Meanwhile, if d_j has smaller values in waiting time, arrival time and price than d_i, we say that d_j *dominates* d_i, and hence d_i should be pruned as it will never make it to the set of skyline drivers.

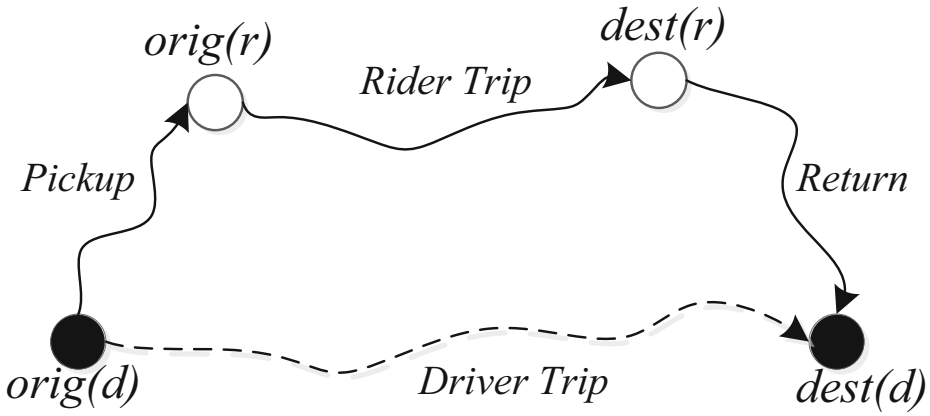


Fig. 2 The illustration for different costs

2.3 Price Cost Model

The main distinguishing point of SHAREK* is its price cost model that is taken into consideration when matching drivers with riders. Figure 2 gives an illustration example for the price cost model of SHAREK*. In this figure, the origin and destinations of driver d and rider r are plotted by black and white circles, respectively. The dotted line represents the original driver d trip from his origin to his destination. The solid line represents the detour that the driver d will encounter to provide a ride sharing service to the rider r. Basically, d has to travel from his origin location $orig(d)$ to the rider origin location $orig(r)$. Then, d has to go through the rider trip till $dest(r)$ to drop off r. Finally, d will need to go to his destination $dest(d)$.

Given the example in Fig. 2, the price for the ride sharing service offered from driver d to rider r, $Price(d, r)$, has two components: (1) The cost of the rider trip from its origin to destination, $RiderTrip(r)$. This is intuitive as at least the rider needs to pay the cost of its own route. Notice that this part is independent from the driver d, i.e., any driver d will be offering ride sharing service to r will include this cost in its price. (2) The cost of the detour that the driver d will encounter to pickup and drop off r, then to return to his own destination. This part of the price cost will play a major role in matching drivers to riders, as drivers with less detour will be favored over drivers with longer detours. Formally, the price can be represented by the following equation:

$$Price(d, r) = RiderTrip(r) + Detour(d, r)$$

$Detour(d, r)$ can be calculated as the difference between the new route of the driver d (the solid line in Fig. 2) and its original route (the dotted line in Fig. 2). This can be formally stated as:

$$\begin{aligned} Detour(d, r) = Pickup(d, r) + RiderTrip(r) \\ + Return(d, r) - DriverTrip(d) \end{aligned}$$

From the above two equations, we get the equation used in SHAREK* to calculate the cost of any ride sharing service from driver d to rider r, as

$$\begin{aligned} Price(d, r) = Pickup(d, r) + 2 * RiderTrip(r) \\ + Return(d, r) - DriverTrip(d) \end{aligned} \quad (1)$$

It is important to note here that the price cost of any trip between two end points is proportional to the shortest path road network distance between the two end points. For example $Pickup(d, r)$ is proportional to the shortest path network distance between $orig(d)$ and $orig(r)$.

2.4 Problem Definition

Based on the understanding of the roles of *Drivers* and *Riders*, along with the price cost model, SHAREK* defines its ride sharing service as follows:

Definition 1 Given a a set of drivers D, where each driver $d \in D$ has a current origin location $orig(d)$ and a destination $dest(d)$, and a ride sharing request from a rider r, located at $orig(r)$ to go to $dest(r)$, within a maximum waiting time $r_{max\_WaitTime}$, a maximum

arrival time $r_{max\_ArrivalTime}$ and a maximum price r_{max_Price}, SHAREK* finds a set of drivers $D' \subset D$, where $\forall d \in D'$, the following hold:

1. $Pickup(d, r) < r_{max\_WaitTime}$,
2. $Pickup(d, r) + RiderTrip(r) < r_{max\_ArrivalTime}$,
3. $Price(d, r) < r_{max\_Price}$,
4. d is in the set of skyline drivers based on $Pickup(d, r)$ and $Price(d, r)$.

It is noted that in the first condition we compare $Pickup(d, r)$, which is a a shortest path distance with $r_{max\_WaitTime}$, which is a time unit. However, this is still accurate as we consider that the shortest path distance between point A and point B is proportional to the time taken to travel from A to B and also to the price to be paid for the trip from A to B. Conversions between distance, time, and price can be done by just multiplying in a factor. Hence, in this paper, we compare distance, time, and price units to each other.

Based on the above conversions, the left side of the second condition, i.e., $Pickup(d, r) + RiderTrip(r)$, can be regarded as the arrival time for the rider r when d is assigned to give a ride to him. Since the $RiderTrip(r)$ is fixed for all the drivers, the arrival time is determined by the value of $Pickup(d, r)$, i.e., the waiting time for r to be picked up by d. As a result, we can combine the first two conditions of Definition 1 to following formula:

$$Pickup(d, r) < min(r_{max\_WaitTime}, r_{max\_ArrivalTime} - RiderTrip(r)) \quad (2)$$

Hence, there is no need to compute skyline drivers from the perspectives of waiting time, arrival time and price, instead, the skyline drivers in our problem can be only defined by two dimensions, namely $Pickup(d, r)$ and $Price(d, r)$. This reduction on skyline dimension is important since it can be utilized to enhance the matching speed as we will see later in SHAREK* query processing.

3 Dynamic Matching in SHAREK*

A naive way to support the ride sharing matching as defined in SHAREK* is to first compute the shortest path between the rider and every single registered driver as well as the shortest path for the driver to return to his original destination after giving the requested ride to the rider. For those drivers that can satisfy both the pickup time and cost constraints, run a two-dimensional skyline algorithm over pickup time and cost to get those drivers that are not dominated by any other drivers. Though the solution looks simple, it has a prohibitive cost of computing large numbers of shortest paths, which is not suitable, given the online environment of ride sharing requests.

SHAREK* avoids such prohibitive cost by deploying a set of early pruning techniques with the goal of minimizing the need for shortest path computations. In fact, we will see that we can efficiently and accurately satisfy the ride sharing request with only few shortest path computations. SHAREK* is composed of three consecutive phases, namely, *Euclidian Temporal Pruning*, *Euclidean Cost Punning*, and *Semi-Euclidean Skyline-aware Pruning*. The three phases are described in details in the rest of this section. For illustration, we use the example shown in Fig. 1 throughout the whole section. We assume that maximum

waiting time, maximum arrival time and maximum price rider constrains are 15, 30 and 30, respectively.

3.1 Data Structure

Driver Table SHAREK* maintains one big table, *Driver Table*, that includes an entry for each currently registered driver with SHAREK*. Once a driver d indicates his willingness to provide a ride sharing service, d is registered with SHAREK* and a new entry for d is added to the *Driver Table* with the following information: A driver entry d has four attribute: (1) *ID*: A unique driver identifier set by SHAREK*, (2) *CurrentLocation*: The current location of d, either set explicitly by d or extracted from her mobile device. With the registration, d allows SHAREK* to track his location, and hence this attribute is continuously changing with the movement of d, (3) *Destination*: The destination location for the driver. Once d reaches to its destination, it is automatically unregistered from SHAREK* and its entry is deleted from the *Driver Table*, and (4) *DriverTrip*: The shortest path cost between *CurrentLocation* and *Destination*. As the *CurrentLocation* is continually changing, the value of the *DriverTrip* changes accordingly. We use an efficient incremental shortest path algorithm [30] for updating the value of *DriverTrip*.

Grid Index The *Driver Table* is indexed by a simple gird index [24] on the *CurrentLocation* field. We opt for using the grid index due to its simplicity and low update overhead. As *CurrentLocation* is a continuously changing fields, it is important to ensure that it does not cause much overhead to the index structure, and hence the grid index is a suitable one.

3.2 Phase I: Euclidean Temporal Pruning

The input of this phase is the set of all registered drivers in the system, stored in the *Driver Table* and the grid index of them. The output is a set of candidate drivers that can pick up the rider r within its maximum waiting time and arrival time, based on Euclidean distance computations. The Euclidean distance between any two points is equal or less than the actual shortest path road network distance between the same two points. Meanwhile, computing the Euclidean distance has a trivial cost compared to the actual shortest path computations. Hence, Euclidean distance can act as a cheap conservative proxy for the actual road network distance.

Main idea The main idea of this phase includes two consecutive steps. In the first step, we identify the real temporal constraint for the rider r based on his requirements of maximum waiting time and maximum arrival time. According to the second condition of Definition 1, we know that the maximum arrival time consists of waiting time and the time spent for the rider trip. Thus, besides the maximum waiting time that explicitly input by the rider, we can also get another possible maximum waiting time constraint implied by the maximum arrival time that the rider required, i.e., subtracting $RiderTrip(r)$ from $r_{max\_ArrivalTime}$. Then, we compare these two values and use the shorter one as the real maximum waiting time constraint. This is because the longer maximum waiting time value will cause the conflict between the two requirements of the rider.

The second step of this phase is to exploit the grid index data structure by a circular range query Q_R centered at the rider location with a radius equivalent to the Euclidean distance

corresponding to the real maximum waiting time constraint of the rider r derived in the first step. Any driver d that does not satisfy the query Q_R is immediately pruned from our consideration, with no further computations, as d will never be able to pick up r within its time constraint. The set of drivers that satisfy the range query Q_R may still include a set of *false positives*, i.e., a driver d in Q_R may still not be able to get to rider r, using the road network distance.

Example Figure 1 gives the temporal pruning result for our running example. Assume that the shortest path cost for the rider trip $RiderTrip(r)$ is 12. Then, the maximum waiting time induced by the maximum arrival time is $r_{max\_ArrivalTime} - 12 = 18$, which is larger than the maximum waiting time required by the rider, i.e., 15. Hence, the real maximum waiting time constraint of r is finalized to 15, and a range query is submitted for retrieving drivers that are within Euclidean distance of the maximum waiting time $r_{max\_WaitTime} = 15$. As a result, nine drivers (i.e., d_1, d_2, ..., d_9) are selected, and these drives' trips are shown in Fig. 3.

3.3 Phase II: Euclidean Cost Pruning

The input to this phase is the set of candidate drivers produced from Phase I, while the output is a subset of the input drivers that are still candidate to be reported in the final answer.

ID	*Pickup*	*Euclidean Pickup*	*Return*	*Euclidean Return*	*Driver Trip*
d_1					*13.4*
d_2					*10*
d_3					*9.6*
d_4					*8.9*
d_5					*9.5*
d_6					*14.8*
d_7					*11.5*
d_8					*12.3*
d_9					*10.7*

Fig. 3 Matching tbale

Main idea The main idea of this phase is to adopt a conservative estimation of the total ride sharing cost (Equation 1), by substituting the road network cost with its corresponding Euclidean cost. Hence, we get the following equation:

$$\begin{aligned} &EuclideanPrice(d,r) = \\ &EuclideanPickup(d,r) + 2 * RiderTrip(r) + \\ &EuclideanReturn(d,r) - DriverTrip(d) \end{aligned} \tag{3}$$

Comparing (1) and (3), as $EuclideanPickup(d,r) < Pickup(d,r)$ and $EuclideanReturn(d,r) < Return(d,r)$, then $EuclideanPrice(d,r)$ must be less than $Price(d,r)$. Hence, if a certain driver cannot satisfy the price constraints, using $EuclideanPrice(d,r)$, then there is no need to calculate any shortest path cost for d, as it will never be able to make it using its road network cost $Price(d,r)$, which is higher than its Euclidean cost.

Algorithm 1 Phases I & II: euclidean temporal & cost pruning.

Input : *Driver Table* with its grid index *g_index*; a rider r
Output: A set of candidate drivers stored in *Matching Table*

$origin(r) \leftarrow$ get the origin of r;
$r_{max\_WaitTime} \leftarrow$ get the maximum waiting time of r;
$r_{max\_ArrivalTime} \leftarrow$ get the maximum arrival time of r;
$RiderTrip \leftarrow$ compute the shortest path cost between $orig(r)$ and $dest(r)$;
if $r_{max\_WaitTime} > 0$ && $r_{max\_ArrivalTime} - RiderTrip > 0$ **then**
  $r_{max\_Time} = \min(r_{max\_WaitTime}, r_{max\_ArrivalTime} - RiderTrip)$;
  Matching Table $\leftarrow Q_R\ (origin(r), r_{max\_Time}, g\_index)$;
  for $d \in$ *Matching Table* **do**
    $EuclideanPickup(d,r) \leftarrow$ compute the Euclidean distance between $orig(d)$ and $orig(r)$;
    $EuclideanReturn(d,r) \leftarrow$ compute the Euclidean distance between $dest(r)$ and $dest(d)$;
    if $EuclideanPickup(d,r) + 2 * RiderTrip(r) + EuclideanReturn(d,r) - DriverTrip(d) > r_{max\_Price}$ **then** `// Equation 2`
      remove d from *Matching Table*;

return A set of candidate drivers stored in *Matching Table*

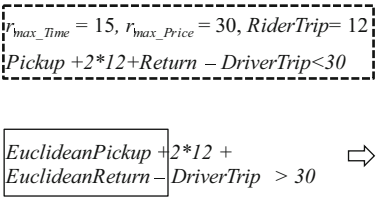
Algorith Algorithm 1 gives the pseudo code for the first two phases. First, we obtain the rider r information, i.e., its location, temporal constraints and the shortest path from $orig(r)$ to $dest(r)$. Then, we combine different temporal constraints to a single one (denoted by r_{max_Time}) based on the (2) (line 5-6), i.e., the maximum waiting time that actually play the role of restricting the driver from picking up the rider. Next, we issue a range query Q_R that exploits the grid index *g_index* over the *Driver Table* to return the set of drivers that can reach to the rider within the temporal constraint using Euclidean distance. The output of the range query is used to populate a newly created table, termed *Matching Table* (Fig. 3), that includes one entry for each driver returned from Phase I. Each driver entry includes the driver id, road network and Euclidean cost for both the pickup and return trip for that driver, and the road network distance of the driver trip. Since the driver trip cost is already known from the driver entry in the *Driver Table*, it is just copied here upon the table initialization. Then, we compute the actual road network cost of the rider trip, which is the shortest path

between its origin and destination. Since we only have one rider, this is a *one time* cost of shortest path query. Then, we scan over all the drivers in the *Matching Table*. For each driver d, we calculate the Euclidean distance for picking up the rider ($EucledianPickup(d, r)$) and returning from the rider destination ($EucledianReturn(d, r)$), and store them in the *Matching Table*. Then, we check if the total cost of driver d (3) is still within the rider cost constraint r_{max_Price}. If not, we exclude d, and remove it from the *Matching Table*, without doing any shortest path computation for d.

Example Figure 4 gives the *Matching Table* after we compute the Euclidean Pickup and Euclidean Return costs for each of the nine drivers d_1 to d_9 that are produced from Phase I, using the actual temporal constraint $r_{max\_Time} = 15$. Assume that the shortest path cost for $RiderTrip(r) = 12$, then we calculate (3) for each of the nine drivers. We find that the total cost for driver d_9 is actually more than 30, which is the maximum price set by the rider r. Hence, we decide to remove driver d_9 from the matching table without any shortest path computation, as we are sure that d_9 will never make it to the final result as its road network distance will exceed its Euclidean distance. The rest of drivers d_1 to d_8 are still candidates as their total cost (using Euclidean distance) is less than 30, hence they still have a chance to make it.

3.4 Phase III: Semi-Euclidean Skyline-aware Pruning

The input to this phase is the set of candidate drivers, produced from Phase II, and stored in the *Matching Table*. The output is the final answer returned to the rider r that includes a set of drivers who not only satisfy the temporal and cost constraints of r, but also represent a set of the skyline result, in terms of time and price, of those drivers that satisfy the temporal and cost constraints. One direct approach to realize this phase is to just calculate all the shortest path (i.e, road network distance) for $Pickup$ and $Return$ for each driver in the *Matching Table*. Then, calculate the actual total cost for each driver in the *Matching Table*. Finally, run a traditional skyline algorithm over total cost and pickup time to get the final answer. Unfortunately, such approach is prohibitively expensive, as it needs to calculate two road network distances ($Pickup$ and $Return$) for each driver in the *Matching Table*, followed by an expensive skyline operation.



ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_1		6.1		13.2	13.4
d_2		6.0		4.2	10
d_3		6.1		9.5	9.6
d_4		4.5		6.6	8.9
d_5		6.5		6.5	9.5
d_6		7.4		8.3	14.8
d_7		9.3		6.8	11.5
d_8		8.7		5.4	12.3
d_9		11.3		9.4	10.7

Fig. 4 Cost pruning with r_{max_Price}

In SHAREK*, we avoid such expensive computations by employing two techniques: (1) We avoid the computations of all pickup and return shortest paths for each driver through early pruning techniques where some drivers can be completely pruned without calculating their shortest paths, and (2) We inject the skyline computations inside the pruning techniques, which helps in pruning even more drivers without any shortest path computations. This achieves a significant improvement as instead of considering the skyline computation as an overhead to be added to our two main constraints (time and cost), we actually consider the skyline operation as a blessing, where we take advantage of it to even prune more drivers without further computations.

Main idea There are actually four main ideas in this phase. First, we use a less conservative Semi-Euclidean equation for computing the total cost than (3). In particular, we use the following equation:

$$\begin{aligned} SemiEuclideanCost(d,r) = \\ Pickup(d,r) + 2 * RiderTrip(r) + \\ EuclideanReturn(d,r) - DriverTrip(d) \end{aligned} \tag{4}$$

Comparing (4) and (3), here we use the actual road network for the pickup cost, while we are still conservative as we still use the Euclidean distance for the return trip. The second idea of this phase is that we retrieve drivers one by one based on their road network pickup distance. This means that if the road network distance of some driver d_i is not satisfying the temporal constraints, then there is no need to continue getting more drivers. The third idea is that we inject the skyline computations in this phase by always setting a maximum cost MAX as the maximum acceptable cost for any driver to be included in the skyline result. MAX is initialized by r_{max_Price}, and is then tightened with every added driver to the final result. The fourth idea is that we sort the *Matching Table* based on the value of $(EuclideanReturn(d,r) - DriverTrip(d))$, which will significantly help in early pruning a set of drivers as will be seen below.

Based on these ideas, we employ an incremental road network nearest-neighbor (INN) algorithm [25] that retrieves the drivers one by one based on their actual road network distance $Pickup(d,r)$ from the rider r. For each driver d, retrieved from the INN query, with an exact road network distance $Pickup(d,r)$ (computed as part of the INN), we will have one of the following four cases:

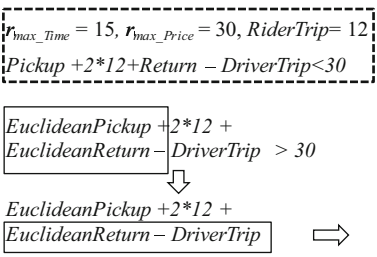
Case 1 Driver d cannot make it on time to pick up the rider, and hence does not satisfy the temporal constraint of rider r, i.e., $Pickup(d,r) > r_{max\_Time}$. In this case, we terminate our algorithm and report the current answer, if any, as the final answer. We do so without the need to calculate the actual road network distance of the return trip nor to calculate any road network distance for the set of drivers that we did not visit yet. The idea is that since driver d cannot arrive punctually, and as we are visiting drivers through an INN algorithm, intuitively all other drivers are further than driver d, and hence none of them will be qualified . Hence there is no need to check any of them.

Case 2 Driver d can satisfy the rider's temporal constraint, i.e., $Pickup(d,r) \leq r_{max\_Time}$. Yet, its semi-Euclidean conservative cost (4) is more than the MAX value. This means that we have visited some driver d_i before with an actual total cost (computed per Equation 1) that is less than the total cost of d. Since d_i is visited before d, then d_i is closer to r than d. Hence, d_i dominates d as its closer to r than d and also it will provide less cost than d.

In this case, we take the following two actions: (1) We consider driver d as not qualified to be in the query answer, even though we did not calculate its actual road network return trip. It is important to note that driver d may still satisfy the cost constraint of the rider, yet, it does not belong to the set of skyline drivers as it is dominated by a prior driver d_i. This is the case where we take advantage of the skyline constraint to early prune drivers without further computation. (2) We prune out all the drivers in the sorted *Matching Table* that are below driver d, i.e., have larger value for (*EuclideanReturn(d,r) - DriverTrip(d)*) than that of driver d, without the need to calculate any road network cost for them. The rational here is that these drivers will have larger values than driver d in $Pickup$ as they are not reported yet using the INN algorithm. Since they will have larger $Pickup$ cost and also larger value of (*EuclideanReturn(d,r) - DriverTrip(d)*). Thus, we can safely prune these drivers as they will never make it to the final skyline answer.

Case 3 Driver d can satisfy the rider's temporal constraint, i.e., $Pickup(d, r) \leq r_{max\_Time}$, and its semi-Euclidean conservative cost (4) is less than the MAX value. Yet, its actual total cost (Equation 1) is more than MAX. Notice that in this case, we had to, for the first time, calculate the actual road network distance of $Return(d, r)$. In this case, we just conclude that driver d cannot make it to the final answer, either because driver d does not satisfy the rider cost constraint or because it will never make it to the skyline answer because of the tightened MAX value. It is important to note here that we cannot prune more drivers from the *Matching Table* as we are using the actual road network distance while the *Matching Table* is sorted based on an Euclidean distance computations.

Case 4 None of the above, which means that d can satisfy the rider's temporal maximum waiting time constraint, its semi-Euclidean conservative cost (4) is less than the MAX value, and its computed actual total cost (1) is less than MAX. In this case, we: (a) add driver d to the final query answer as we conclude that d is a qualified driver who belong to the set of skyline drivers that can pick rider r within its time and cost constraints, and (b) tighten the value of MAX to be the total cost of d. Such tightening is important as it indicates that for any other driver d_i to be reported in the final answer, d_i has to have less cost than that of d to be a skyline. Notice that d_i will definitely have higher $Pickup$ cost than that of d as we are retrieving drivers using an INN algorithm. So, to be in a skyline, driver d' must have a ride sharing cost that is less than that current driver d being iterated.



ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_8		8.7		5.4	12.3
d_6		7.4		8.3	14.8
d_2		6.0		4.2	10
d_7		9.3		6.8	11.5
d_5		6.5		6.5	9.5
d_4		4.5		6.6	8.9
d_1		6.1		13.2	13.4
d_3		6.1		9.5	9.6

Fig. 5 *Matching Table* sorting based on $(EuclideanReturn(d, r) - DriverTrip(d))$

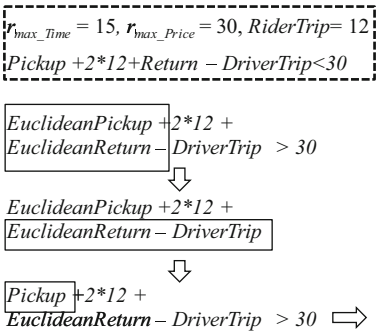
Algorithm 2 Semi-euclidean skyline-aware pruning.

Input : A set of drivers generated from Euclidean distance pruning and a rider r
Output: skyline drivers R

calculate $(EuclideanReturn(d,r) - DriverTrip(d))$ for each driver d in *Matching Table*;
sort *Matching Table* by values of $(EuclideanReturn(d,r) - DriverTrip(d))$ in an ascending order;
$MAX \leftarrow r_{max\_Price}$;
while *Matching Table* $\neq \emptyset$ **do**
  $Pickup(d,r) \leftarrow$ retrieve the nearest driver d in *Matching Table* from $orig(r)$; `// INN query`
  if $Pickup(d,r) > r_{max\_Time}$ **then**
    break ; `// terminate the program`
  if $Pickup(d,r)+2*RiderTrip(r)+EuclideanReturn(d,r)-DriverTrip(d) > MAX$ **then** `// Equation 3`
    remove d and drivers that below d from *Matching Table*;
  else
    $Return(d,r) \leftarrow$ compute the shortest path cost between $dest(r)$ and $dest(d)$;
    if $Pickup(d,r)+2*RiderTrip(r)+Return(d,r)-DriverTrip(d) > MAX$ **then** `// Equation 1`
      remove d from *Matching Table*;
    else
      $MAX \leftarrow Pickup(d,r) + 2 * RiderTrip(r) + Return(d,r) - DriverTrip(d)$;
      $R \leftarrow$ move d from *Matching Table* to the result set;
return skyline driver set R

Algorithm Algorithm 2 gives the pseudo code for Phase III. We first sort the *Matching Table* based on the value of $(EuclideanReturn - DriverTrip)$, computed for each driver. Then, we initialize a MAX value to the maximum price of the rider, i.e., r_{max_Price}. Next, we iterate over the sorted *Matching Table*. For each iteration, we execute the nearest-neighbor query to retrieve the driver d with the lowest road network cost of $Pickup$. If d cannot satisfy the rider temporal constraint, we conclude by reporting the current set of skyline drivers. Otherwise, we calculate the semi-Euclidean cost of d (4). If it is more than the current value of MAX, we shrink the *Matching Table* by removing d along with all drivers below d. On the other side, if the semi-Euclidean cost of d is less than MAX, we have to calculate the actual road network cost for the return trip of d (line 11). Then, we calculate the actual total cost of d (line 12). If such cost is still more than MAX, we just remove d only from the *Matching Table*, otherwise, we add d to the final result, and update the value of MAX accordingly.

Example We continue our running example, where rider r, with road network distance trip 12, needs to be picked up within the constrains $r_{max\_Time} = 15$ and $r_{max\_Price} = 30$. Figure 5 gives the *Matching Table* sorted based on $(EuclideanReturn - DriverTrip)$



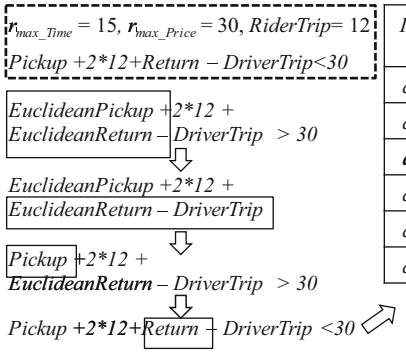
ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_8		8.7		5.4	12.3
d_6		7.4		8.3	14.8
d_2	7	6.0		4.2	10
d_7		9.3		6.8	11.5
d_5		6.5		6.5	9.5
d_4		4.5		6.6	8.9

ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_8		8.7		5.4	12.3
d_6		7.4		8.3	14.8
d_2		6.0		4.2	10
d_7		9.3		6.8	11.5
d_5		6.5		6.5	9.5
d_4		4.5		6.6	8.9
d_1	6.5	6.1		13.2	13.4
d_3		6.1		9.5	9.6

Fig. 6 Filtering based on Equation 3

in an ascending order. Then, *Semi-Euclidean Distance Pruning* will iterate over drivers by querying incremental nearest neighbor. Figure 6 gives the procedure for filtering based on the estimation described in Eq. 4. First, INN query retrieves the nearest driver d_1. We find that the cost of d_1 is more than the rider constraint ($6.5 + 2 * 12 + 13.2 - 13.4 > 30$), hence driver d_1 is removed from *Matching Table*. In addition, based on the conclusion of *Matching Table* sorting, driver d_3 is also filtered out without any shortest path computation.

Figure 6 also gives the next iteration, where we can see that after filtering out d_1 and d_3 (shown in grey area of the lower table), the *Matching Table* is shrunk to be the table in the top right corner. Then, we continue to search the nearest driver in this shrunk table, d_2 is chosen, where its $Pickup$ cost is not only within the real maximum waiting time ($r_{max\_Time} = 15$), but also it has a possible ride sharing cost less than $r_{max\_Price} = 30$. So, we calculate the shortest path cost of $Return$ for d_2. As shown in Fig. 7, we use the



ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_8		8.7		5.4	12.3
d_6		7.4		8.3	14.8
d_2	**7**	**6.0**	**8.3**	**4.2**	**10**
d_7		9.3		6.8	11.5
d_5		6.5		6.5	9.5
d_4		4.5		6.6	8.9

Fig. 7 Filtering based on Cost Constraint

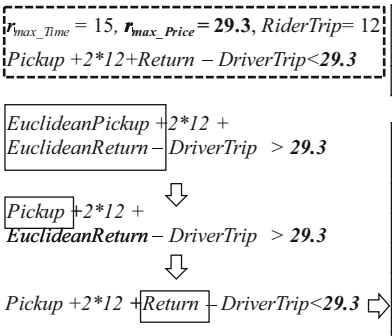
ride sharing cost constraint (1) shown in the left bottom of the figure to test d_2. Fortunately, d_2 can meet the demand of the rider's maximum price, i.e., d_2 provides a price of 29.3 $(9.1 + 2 * 12 + 7.6 - 10)$, which is less than 30. As a result, d_2, denoted by the bold font in right table, is retained in the *Matching Table*, and since d_2 passes the last evaluation of cost constraint (1), driver d_2 becomes a qualified one.

Figure 8 describes the skyline processing case. The driver d_2 is the first found qualified driver with ride sharing cost 29.3, which is less than current MAX (30), and the time for picking up is 7 which must be the least waiting time among all the drivers according to the INN algorithm we are using. Since the next found skyline driver must cause more waiting time for the rider, then the ride sharing cost of this prospective driver must be less than 29.3. Thus, to find this driver in the next iteration, we update the maximum price/ride sharing cost r_{max_Price} to 29.3 as shown in the top left part of the figure. Based on this new value, we filter out driver d_5 when evaluating (1) $(8.2 + 2 * 12 + 8 - 9.5 > 29.3)$. Similarly, drivers d_4, d_8, d_6, and d_7 are pruned through using (4) or (1) in following iterations. Eventually, only one driver d_2 out of 10 is returned to the rider r.

3.5 Materialization in SHAREK*

As introduced in Algorithm 2, SHAREK* adopts both the incremental nearest neighbor (INN) query (line 5) and the shortest path query (line 11) to obtain the road network distance between the rider and the driver when it is needed. To further speed up the dynamic matching, the technique of materialization [7] for retrieving the road network distance is also implemented in SHAREK*. Specifically, there exists two types of materialization:

1. Materialization in the INN query. Whenever the INN query is issued, the nodes and edges of the road network that have been visited will be recorded as the expansion starting point for the next INN query. Hence, SHAREK* won't need to search the nearest driver from scratch on the road network graph, which can greatly accelerate the INN query performance.
2. Materialization in the shortest path query. SHAREK* uses the shortest path query for computing the road network distance between the destinations of the rider and the driver. The shortest path algorithm, e.g., Dijkstra algorithm, may retrieve the network distance for another driver's destination before obtaining the shortest path for the given



ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_2	**7**	**6.0**	**8.3**	**4.2**	**10**

ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_8	9	8.7	10.2	5.4	12.3
d_6	11.1	7.4	9.7	8.3	14.8
d_2	**7**	**6.0**	**8.3**	**4.2**	**10**
d_7	12	9.3		6.8	11.5
d_5	8.2	6.5	8	6.5	9.5
d_4	8.7	4.5	7.5	6.6	8.9

Fig. 8 Skyline processing

driver. In this case, SHAREK* would record all the found network distances for corresponding drivers, i.e., set the values of $Return$ field in the *Matching Table*, which then can be directly used for testing the price cost constraint (1) without issuing the shortest path query. Moreover, the visited nodes and edges of the road network, like the first materialization in INN query, are also marked for continuing the future shortest path search.

4 Dual-sides Pruning

In previous implementation of Phase 3 in SHAREK, i.e., Semi-Euclidean Skyline-aware Pruning Phase, the actual road network distance for $Return$ trip had to be computed once the driver satisfies the rider's temporal constraint and its semi-Euclidean cost is less than the tightened price cost (MAX). It is important to note here that only one driver could be filtered out each time when the final ride sharing cost offered by this driver is greater than the MAX value, which leads to inefficiency when a large number of drivers' $Return$ trip and final cost need to be checked. Hence, in this section, we present an alternative implementation for Phase 3. Unlike previous implementation where the pruning technique started from the $Pickup$ side, we propose a new pruning strategy from both $Pickup$ and $Return$ sides in this new version, which may provide more capability for pruning the search space than previous strategy.

Main idea The new implementation distinguishes itself from its predecessor in following two main aspects. First, to filter out those drivers who are not able to provide ride sharing costs within the rider's cost constraint, we adopt a new Semi-Euclidean equation:

$$\begin{aligned} SemiEuclideanCost(d,r) &= \\ EuclideanPickup(d,r) &+ 2 * RiderTrip(r) + \\ Return(d,r) &- DriverTrip(d) \end{aligned} \tag{5}$$

Comparing (5) and (4), here we use the Euclidean distance for the pickup trip while road network distance for the return trip, which is the opposite of Eq. 4 but is still conservative as only return trip is in the form of road network distance. Second, to prune as many drivers as possible, we separate the semi-Euclidean cost evaluation from the skyline-aware pruning procedure.

Specifically, we firstly remove those drivers whose semi-Euclidean cost (5) can not make it to satisfy the rider's maximum price constraint by incrementally retrieving the driver that has the shortest road network distance of return trip, i.e., $Return(d,r)$. It is important to note here that we are possible to prune more than one drivers whenever the return trip $Return(d,r)$ is evaluated, given that *Matching Table* had been sorted by the values of $(EuclideanPickup(d,r) - DriverTrip(d))$. After this step, the *Matching Table* is shrank and to early prune a set of drivers for following skyline-aware pruning procedure, we turn to the second sorting on *Matching Table* based on $(Return(d,r) - DriverTrip(d))$. Then, we iteratively get drivers from the $Pickup(d,r)$ side by using INN algorithm on road network. The most distinguishable point in this procedure is that during each evaluation against the $Pickup(d,r)$, i.e., using (1), more than one driver could be possibly removed from the *Matching Table* which potentially reduces the effort for calculating road network distance. As a result, compared with the previous Semi-Euclidean Skyline-aware Pruning strategy mentioned in Section 3.4, the new matching design from dual sides of $Pickup$ and $Return$

is more effective in terms of pruning out those drivers who are not able to give rides to the rider.

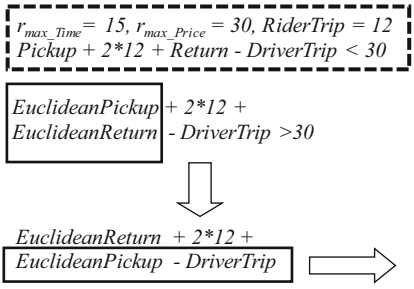
Algorithm 3 Dual-sides pruning for Phase III.

Input : A set of drivers generated from Euclidean distance pruning and a rider r
Output: skyline drivers R

calculate $(EuclideanPickup(d,r) - DriverTrip(d))$ for each driver d in *Matching Table*;
sort *Matching Table* by values of $(EuclideanPickup(d,r) - DriverTrip(d))$ in an ascending order;
while *there exists unprocessed driver in Matching Table* **do**
  $Return(d,r) \leftarrow$ retrieve the nearest driver d in *Matching Table* from $dest(r)$;
  `// INN query`
  if
  $(EuclideanPickup(d,r)+2*RiderTrip(r)+Return(d,r)-DriverTrip(d) > r_{max\_Price}$ **then** `// Equation 3`
    remove d and drivers that below d from *Matching Table*;
sort *Matching Table* by values of $(Return(d,r) - DriverTrip(d))$ in an ascending order;
$MAX \leftarrow r_{max\_Price}$;
while *Matching Table* $\neq \emptyset$ **do**
  $Pickup(d,r) \leftarrow$ retrieve the nearest driver d in *Matching Table* from $orig(r)$;
  `// INN query`
  if $Pickup(d,r) > r_{max\_Time}$ **then**
    break ; `// terminate the program`
  if $Pickup(d,r) + 2 * RiderTrip(r) + Return(d,r) - DriverTrip(d) > MAX$
  then `// Equation 1`
    remove d and drivers that below d from *Matching Table*;
  else
    $MAX \leftarrow Pickup(d,r)+2*RiderTrip(r)+Return(d,r)-DriverTrip(d)$;
    $R \leftarrow$ move d from *Matching Table* to the result set;
return skyline driver set R

Algorithm Algorithm 3 gives the pseudo code for Dual-sides Pruning which is an alternative implementation for Semi-Euclidean Skyline-aware Pruning, i.e., Phase III. In the beginning, we sort the *Matching Table* by values of $(EuclideanPickup(d,r) - DriverTrip(d))$ in an ascending order. Then we iterate the sorted *Matching Table*. In each iteration, we issue an INN query from the rider's destination $dest(r)$ to get the driver that has the shortest return trip. Next we check the retrieved driver d by his semi-Euclidean cost (5). If it is greater than the r_{max_Price}, driver d and the drivers that below d in *Matching Table* will be removed.

After this semi-Euclidean cost pruning step, we move on to the skyline-aware pruning step by sorting the *Matching Table* by values of $(Return(d,r) - DriverTrip(d))$ and initializing a variable MAX with the rider's maximum price r_{max_Price}. Then, similar to the *Matching Table* iteration described in the Algorithm 2, we also use an INN algorithm to retrieve the nearest driver d that can pick up the rider. If the distance $Pickup(d,r)$ is greater



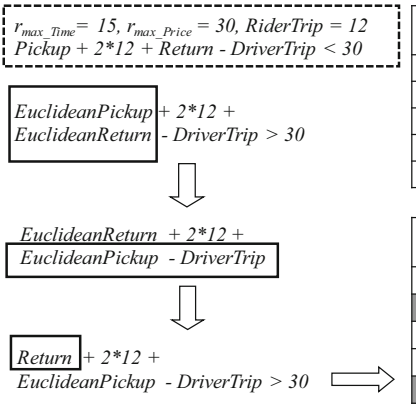
ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_6		7.4		8.3	14.8
d_1		6.1		13.2	13.4
d_4		4.5		6.6	8.9
d_2		6.0		4.2	10
d_8		8.7		5.4	12.3
d_3		6.1		9.5	9.6
d_5		6.5		6.5	9.5
d_7		9.3		6.8	11.5

Fig. 9 *Matching Table* sorting based on $(EuclideanPickup(d, r) - DriverTrip(d))$

than the r_{max_Time}, we terminate the program by returning the current skyline drivers. Otherwise, we calculate the final ride sharing cost offered by the driver d (1). If it is more than the current value of MAX, the *Matching Table* will be shrank again by removing d along with all drivers below d. Otherwise, d will be considered as a qualified driver and moved to the final skyline result, and MAX is also updated to the value of d's ride sharing cost.

Example For the purpose of comparison, we use the same example from Phase III in Section 3.4. The rider's road network distance, time and cost constraints remain the same, i.e., $RiderTrip = 12$, $r_{max\_Time} = 15$ and $r_{max\_Price} = 30$. Figure 9 shows the sorted *Matching Table* in an ascending order of $(EuclideanPickup(d, r) - DriverTrip(d))$. Then an INN query will be executed to retrieve the drivers one by one according to their actual road network distance $Return(d, r)$.

Figure 10 gives the procedure for semi-Euclidean cost pruning based on Eq. 5. First, we get driver d_4 with a $Return$ value of 7.5 and his Semi-Euclidean cost is less than the rider constraint $(4.5 + 2 * 12 + 7.5 - 8.9 = 27.1 < 30)$, hence, d_4 is retained in *Matching*



ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_6		7.4	9.7	8.3	14.8
d_4		4.5	7.5	6.6	8.9
d_2		6.0	8.3	4.2	10
d_5		6.5	8.0	6.5	9.5
d_7		9.3	7.8	6.8	11.5

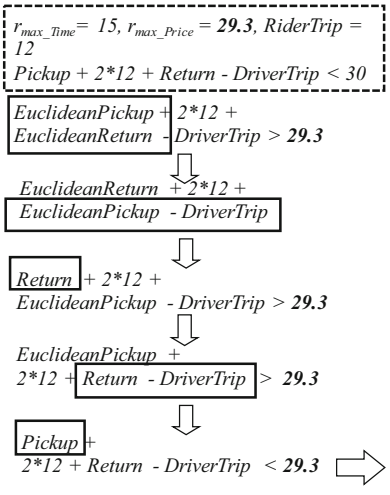
ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_6		7.4	9.7	8.3	14.8
d_1		6.1	15	13.2	13.4
d_4		4.5	7.5	6.6	8.9
d_2		6.0	8.3	4.2	10
d_8		8.7	10.2	5.4	12.3
d_3		6.1		9.5	9.6
d_5		6.5	8.0	6.5	9.5
d_7		9.3	7.8	6.8	11.5

Fig. 10 Semi-Euclidean cost pruning based on Eq. 5

Table. Then, similarly, d_7, d_5, d_2, d_6 are found with $Return$ 7.3, 8.0, 8.3, 9.7 and Semi-Euclidean cost 29.6, 29, 28.3, 27.1 which are all able to satisfy the rider cost constraint and therefore they are all reserved in *Matching Table*. Next, driver d_8 is retrieved with the Semi-Euclidean cost 30.6 ($8.7 + 24 + 10.2 - 12.3$) which is more than the rider cost constraint 30. As a result, d_8 denoted in grey area will be removed from *Matching Table*. Additionally, considering *Matching Table* is sorted, d_3 can be also filtered out without any shortest path computation. In this way, totally three drivers d_1, d_3, d_8 denoted in the grey area of lower table are removed, and the *Matching Table* is shrunk to be the table in the top right corner.

Figure 11 gives the procedure for skyline-aware processing step in Algorithm 3. First of all, drivers in *Matching Table* shown in the bottom left part of of the figure are sorted based on the value of $Return - DriverTrip$. d_2 is the first driver retrieved by employing INN algorithm on $Pickup$ trip, and his actual total cost satisfies the rider's cost constraint ($7 + 2 * 12 + 8.3 - 10 = 29.3 < 30$). Then, d_2 will be moved to final result set. In the meantime, as shown in the top left part of Fig. 11, the maximum price r_{max_Price} is updated to 29.3 because the next found driver must be a skyline driver in terms of the time and the cost. Next, d_5 is found with the cost 30.7 ($8.2 + 2 * 12 + 8 - 9.5$) which is greater than the tightened cost 29.3, hence d_5 is removed. Note that, the driver that below d_5, i.e., d_4 is also removed without any shortest path calculation at $Pickup$ side. Similarly, by computing only one $Pickup$ roadnetwork distance, d_6 and d_7 in grey area of the middle right table are pruned out, and finally, only d_2 is reported to the rider.

Contrast Now we compare Algorithm 2 with Algorithm 3 in terms of the pruning effectiveness. In our example, as shown in Figs. 8 and 11, both two algorithms got the same final answer, i.e., d_2. However, Algorithm 3, i.e., Dual-sides Pruning strategy for SHAREK Phase 3, cost less amount of computation for shortest path than that of Algorithm 2. Specifically, in Algorithm 3, we have totally computed 10 shortest paths where 7 $Return$ trips for d_4, d_7, d_5, d_2, d_6, d_8, and d_1, 3 $Pickup$ trips for d_2, d_5 and d_6. Meanwhile, to get the final answer of d_2, Algorithm 2 requires 12 shortest path calculations, i.e., 7 $Pickup$ trips for d_1, d_2, d_5, d_4, d_8, d_6, and d_7, 5 $Return$ trips for d_2, d_5, d_4, d_8, and d_6. As a result, the



ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_2	7	**6.0**	**8.3**	**4.2**	**10**

ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_6	11.1	7.4	9.7	8.3	14.8
d_7		9.3	7.8	6.8	11.5
d_2	7	6.0	8.3	4.2	10

ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_6		7.4	9.7	8.3	14.8
d_7		9.3	7.8	6.8	11.5
d_2	7	6.0	8.3	4.2	10
d_5	8.2	6.5	8.0	6.5	9.5
d_4		4.5	7.5	6.6	8.9

Fig. 11 Skyline-aware pruning based on Eq. 1

Dual-sides Pruning strategy in Algorithm 3 can provide a better pruning effect as well as the query performance. The experiments later also prove this observation.

5 Experimental Evaluation

This section provides experimental evaluation of SHAREK* based on an actual system implementation. We first compare the overall performance of SHAREK* (Section 5.1), then we investigate the performance of each phase in SHAREK* (Section 5.2). All Experiments in this section are based on a mixture of real and synthetic data sets. The real part comes from the road network of Los Angles, CA, USA, containing 530,977 edges and 193,948 nodes. The synthetic data set are the drivers and riders on the road network, which are generated according to Brinkhoff road network generator [27]. In our experiment, we consider 1,000 rider requests, where we report the average performance for all these requests. We assume the average speed for each driver is 60 km per hour, the ride sharing cost is one dollar per KM. Drivers are indexed by a grid index where the side length of each grid cell is 0.01 degree of longitude by 0.01 degree of latitude, which corresponds to a road network distance of 1KM around. The incremental nearest neighbor (INN) algorithm is implemented based on the main idea of incremental network expansion (INE) [25]. All experiments are evaluated on a server machine with Intel(R) Xeon(R) CPU E5-2603 v2 @1.8 GHz processor and 32 GB RAM with Ubuntu Linux 14.04.

5.1 Overall Performance

This section studies the average response time of SHAREK*. Our previous study on SHAREK [6] had shown that an exhaustive search by computing the shortest path cost for all drivers is the most inefficient method and is impractical for dynamic ride sharing scenario. Hence, we ignore this pure shortest path computation method as the competitor. Moreover, considering that the maximum waiting time and maximum arrival time of the rider can be converted to each other, we use the maximum waiting time as the only temporal constraint for the rest of our experiment.

In Fig. 12, we compare SHAREK* with its predecessor SHAREK from the perspective of whether skyline processing is embeded: (1) SHAREK*-DP is a variant of SHAREK*, where we use the dual-sides pruning strategy mentioned in Section 4 to implement the Phase III of SHAREK*, i.e., Semi-Euclidean Cost Pruning phase. (2) Each competitor appended with *-NoSkyline* represents its corresponding variant where we use all the pruning techniques we have, except skyline pruning. We vary the maximum waiting time from 5 to 60 minutes, while fixing the driver number to 100,000 and the maximum price to 5 dollars. In order to avoid skewness towards large values of long matching time, all experiments in Fig. 12 are plotted with a logarithmic scale of base 10.

As we can see from the figure, both SHAREK* and SHAREK*-DP clearly outperform SHAREK by more than 2 orders of magnitude in terms of the average response time. Such significant performance gain can be ascribed to the materialization on INN and shortest path search. In addition, combining with the results shown in Table 1, we find that three SHAREKs (SHAREK, SHAREK* and SHAREK*-DP) generally require less response time than their corresponding variants where no skyline processing is embeded, i.e., SHAREK-NoSkyline, SHAREK*-NoSkyline and SHAREK*-DP-NoSkyline. This shows that the skyline functionality to the user constraints does not result in any extra overhead. Instead,

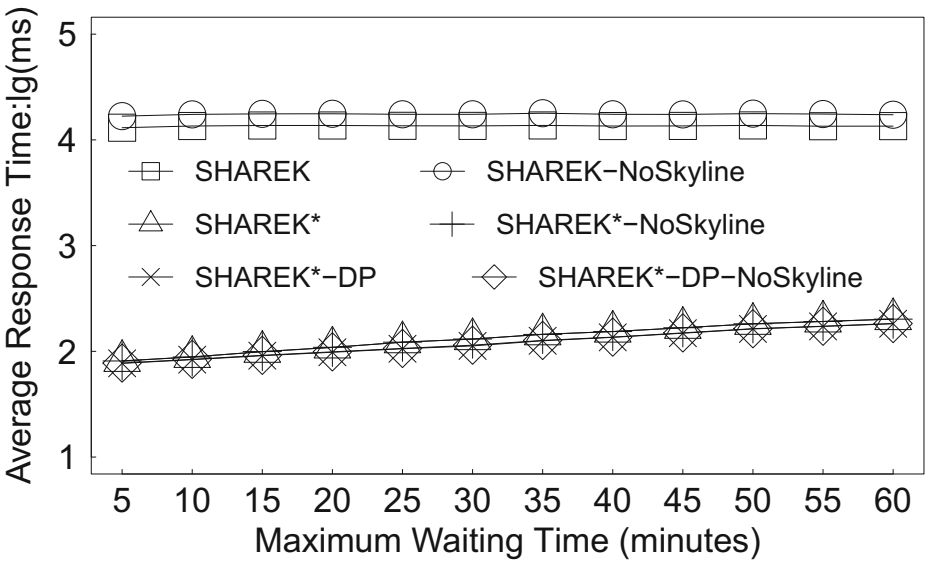


Fig. 12 Overall performance comparison for SHAREK* and its predecessor

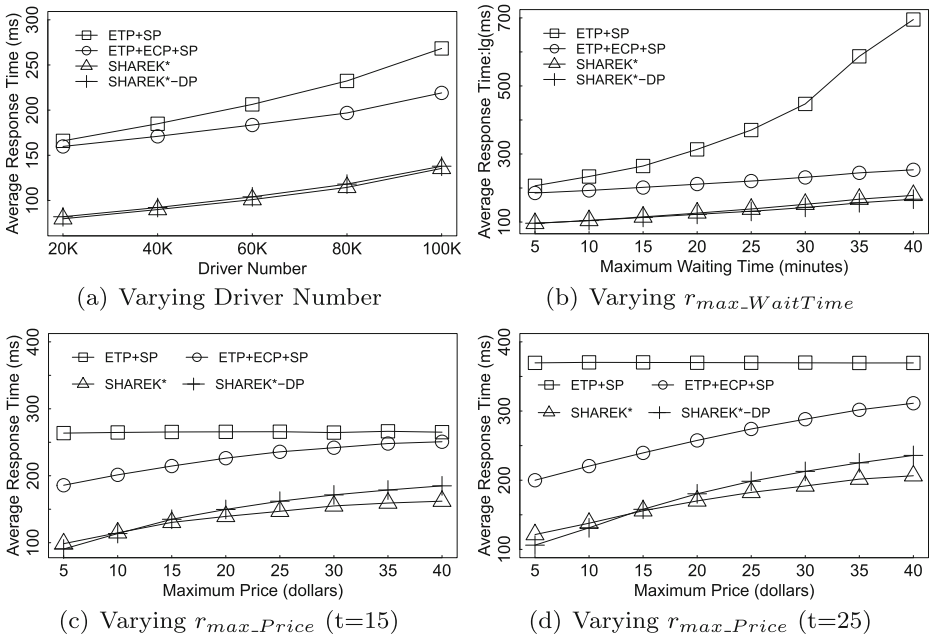
SHAREKs take advantage of the skyline functionality to increase the performance as well as reporting more meaningful answer, i.e., less number of drivers.

Furthermore, as described in Table 1, it is interesting to see that SHAREK*-DP performs worse than SHAREK* when maximum waiting time is within 10 minutes and maximum price r_{max_Price} is set to 15 dollars. The performance of SHAREK*-DP is getting better and better as the maximum waiting time increases. This shows that the dual-sides pruning strategy of SHAREK*-DP needs some extra effort (mainly caused by the sorting) for filtering out some drivers and this overhead exists when the maximum waiting time is small. However, when the waiting time increases many drivers could be possibly answered, and the time which might be spent on pruned drivers offset the pruning overhead within SHAREK*-DP.

In Fig. 13, based on the main idea of dynamic matching in SHAREK*, we compare SHAREK* and SHAREK*-DP with the following two variants: (1) *ETP*+*SP*, i.e., Euclidean Temporal Pruning plus shortest path, where only the first phase of SHAREK* is utilized, followed by computing the shortest path for the rest of drivers to get the final result,

Table 1 Skyline vs No Skyline (against 100,000 drivers; time measurement: milliseconds)

$r_{max\_WaitTime}$	5 mins	10 mins	15 mins	20 mins
SHAREK* ($r_{max\_Price} = 5$)	81.264	88.632	99.099	108.887
SHAREK*-NoSkyline ($r_{max\_Price} = 5$)	81.084	88.403	99.490	109.179
SHAREK*-DP ($r_{max\_Price} = 5$)	77.010	83.387	90.965	98.031
SHAREK*-DP-NoSkyline ($r_{max\_Price} = 5$)	77.471	84.035	98.891	106.574
SHAREK* ($r_{max\_Price} = 15$)	105.640	116.026	127.128	142.283
SHAREK*-NoSkyline ($r_{max\_Price} = 15$)	103.531	116.236	127.990	143.596
SHAREK*-DP ($r_{max\_Price} = 15$)	108.552	116.449	127.074	140.451
SHAREK*-DP-NoSkyline ($r_{max\_Price} = 15$)	108.106	119.284	130.871	145.046



(a) Varying Driver Number

(b) Varying $r_{max\_WaitTime}$

(c) Varying r_{max_Price} (t=15)

(d) Varying r_{max_Price} (t=25)

Fig. 13 Comparison study for SHAREK*

(2) *ETP+ECP+SP*, i.e., SHAREK* with Euclidean Temporal and Cost Pruning, which is basically the first two phases of SHAREK*, followed by shortest path computations of all candidate drivers out of Phase II.

In Fig. 13a, we vary the number of drivers from 20,000 to 100,000, while fixing the rider waiting time constraints to 15 minutes and the maximum price to 15 dollars. Since the *ETP+ECP+SP* method can prune more drivers with the help of Euclidean cost pruning technique, it can get final result faster than *ETP+SP*. However, both *ETP+SP* and *ETP+ECP+SP* methods have unacceptable performance as two orders of magnitude slower than SHAREK* and SHAREK*-DP. This is also due to the need of computing large number of shortest path operations, as the Euclidean-based pruning phases are not enough to ensure an acceptable performance.

In Fig. 13b, we vary the maximum waiting time from 5 to 40 minutes, while fixing the number of drivers to 100,000 and the maximum price to 15 dollars. Performance comparison among the four techniques follow the same trends as that of Fig. 13a. Compared with *ETP+SP*, the performance of all other three methods show a slight upward trend, which is not sensitive to the increase in the waiting time. This is because the maximum price here results in more pruning than the waiting time. In the meantime, SHAREK* and SHAREK*-DP share almost the same performance which is up to 100 ms faster than that of *ETP+ECP+SP*. This shows the pruning capability of SHAREK* and SHAREK*-DP. Moreover, the performance for SHAREK* and SHAREK*-DP goes worse with the increase of the waiting time constraint. This is because the larger the maximum waiting time, the less drivers can be pruned. Hence, more overhead is imposed on the third phase of SHAREK* and SHAREK*-DP.

In Fig. 13c and d, we vary the maximum price from 5 to 40 dollars, while fixing the number of drivers to 100,000 and the waiting time to 15 minutes (t=15) and 25 minutes (t=25)

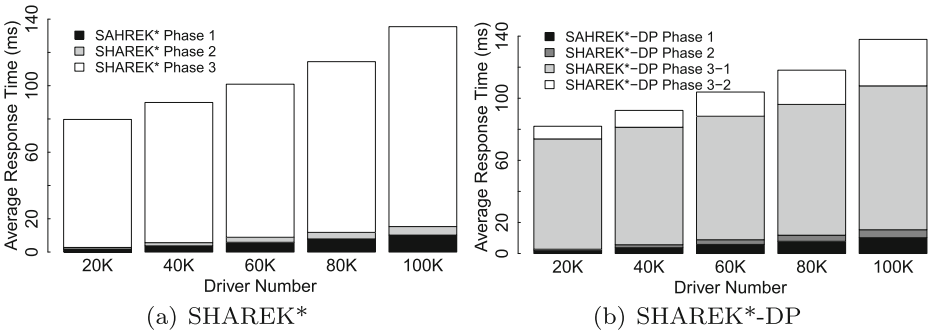


Fig. 14 Response time under different driver number

respectively. Generally, the difference in performance among various algorithms remains the same as in Fig. 13a and b. However, *ETP+SP* shows a stable performance that is not affected by the increase in the price. This is because the the Euclidean temporal pruning phase does not utilize any price based evaluation for filtering drivers. Hence, no matter how much we increase the price, we will still end up with the same number of drivers pruned by the maximum waiting time, and hence the performance is stable. The other three methods all show an increasing trend as the price goes up. Note that, it is interesting to find in both figures that SHAREK*-DP consumes less response time than that of SHAREK* when the price is within 15 dollars. This is because when the price is within 15 dollars, the size of the Matching Table after first semi-Euclidean pruning of SHAREK*-DP is relatively small. As a result, sorting is lightweight and the sorted Matching Table is helpful to prune many drivers in the second semi-Euclidean pruning. But as the price goes up, the size of the Matching Table becomes bigger and bigger, which results in the second sorting consume much time. Therefore, there is a tradeoff between dual-sides pruning and semi-Euclidean pruning.

5.2 Inside SHAREK

This section studies the internals of SHAREK* and its alternative implementation SHAREK*-DP in terms of the performance and pruning power of each phase separately.

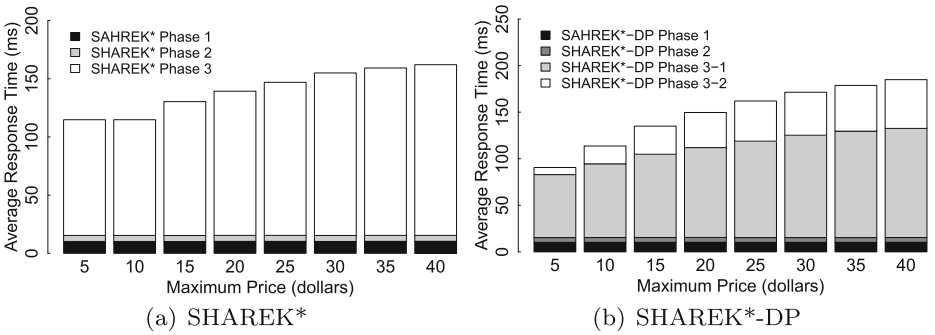


Fig. 15 Response time under different maximum waiting time

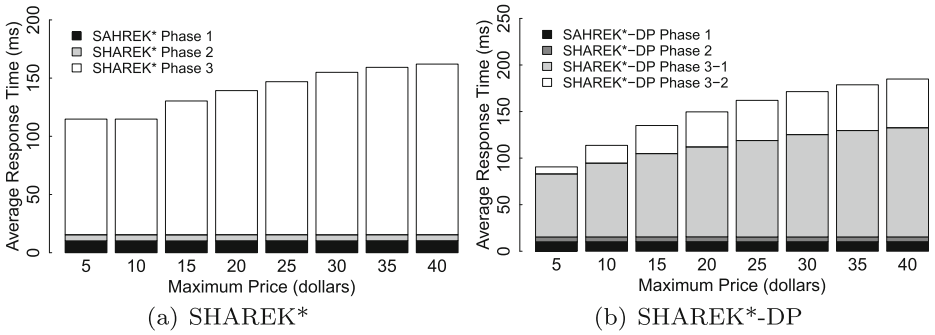


Fig. 16 Response time under different maximum price

SHAREK-DP Phase 3-1* and *SHAREK*-DP Phase 3-2* denote the first and second step of Phase III within SHAREK*-DP respectively.

5.2.1 Response Time for Each Phase

Figures 14, 15 and 16 give the breakout of response time for the three phases of SHAREK* and SHAREK*-DP, when increasing number of drivers, maximum waiting time, and maximum price. The maximum waiting time, maximum price, and number of drivers is set to 15 minutes, 15 dollars and 100,000 drivers. It is clear to see from these figures that Phase III of both SHAREK* and SHAREK*-DP consumes the largest portion of the total response time. This is because the third phase is the only one that needs to make actual shortest path computations in addition to the skyline pruning. Furthermore, the first step of Phase III of SHAREK*-DP cost more time than the second step, this is because many drivers would be visited for computing the shortest path to perform the semi-Euclidean pruning. Meanwhile, less number of drivers, maximum waiting time and shrinking maximum price constraints all work together to guarantee the efficiency of the second step.

In Fig. 14, it is clear that each phase of both SHAREK* and SHAREK*-DP will cost more time as the driver number increases. However, the time consumed for Euclidean pruning is quite small and we can hardly distinguish between the time consumed in Phases I and II compared to Phase III when the driver number is small, i.e., 20,000. This is expected as the first two phases do not encounter any shortest path computations. Figure 15 also

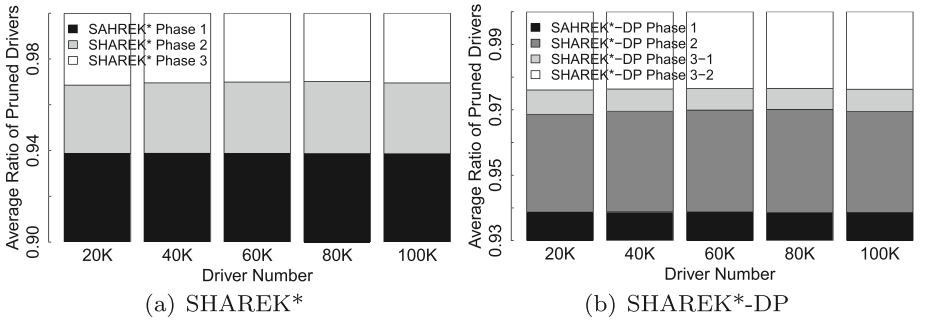


Fig. 17 Average Ratio of Pruned Drivers under different driver number

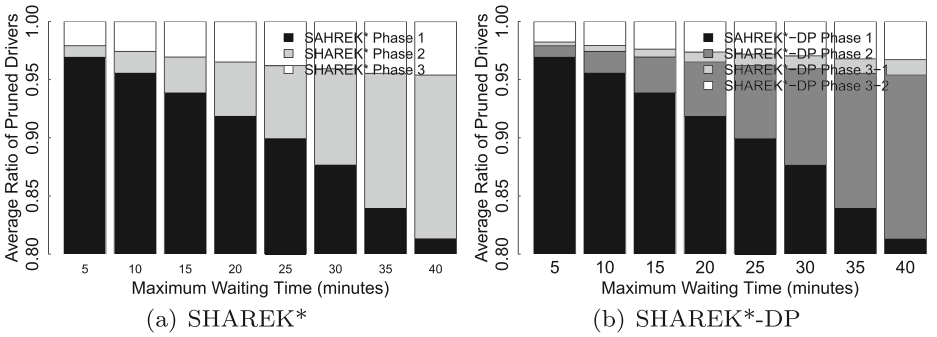


Fig. 18 Average Ratio of Pruned Drivers under different maximum waiting time

shows an upward trend for each phase, the increase of maximum waiting time can finally cause significant influences on the response time of both SHAREK* and SHAREK*-DP. However, it is interesting see from Fig. 16 that the time consumption of Phase I and II in both methods are stable no matter how much price we increase. The main reason behind this is that as long as the temporal constraint is fixed, the number of driver for evaluation is also fixed for Phase I and II, and the computation for each evaluation is more or less the same.

5.2.2 Pruning Ratio for Each Phase

Figures 17, 18 and 19 give the pruning capability of each phase separately. The settings of the experiments are the same as of Figs. 14, 15 and 16. Phase I clearly has the most pruning capability with more than 93.8% in Figs. 17 and 19, more than 81.3% in Fig. 18. In Fig. 17, the pruning ratio for each phase does not change much as the driver number goes up, this is because the distribution of different driver set is similar. Though the driver number is increased, more drivers would be pruned out and more drivers could be retained as well.

In Fig. 18, we find that the pruning ratio for Phase II and III is growing while Phase I is decreasing as the maximum waiting time goes up. This is due to the reason that more drivers could satisfy the Euclidean temporal constraint when we increase the waiting time. Hence, the pruning ratio for Phase I shows a downward trend. In the meantime, more drivers are left for Phase II and III for further pruning and their pruning ratio will increase. In Fig. 19, Phase I has a stable pruning ratio trend which is not affected by the increasing price. The

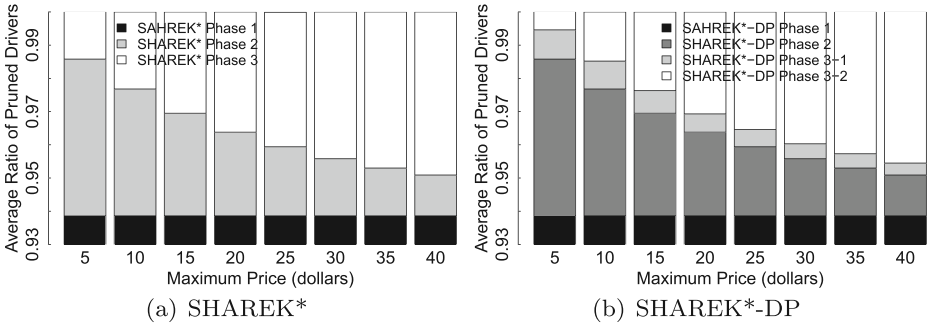


Fig. 19 Average Ratio of Pruned Drivers under different maximum price

reason for this is as same as that of Fig. 16. Different from the case shown in Fig. 18, due to the increasing price which allow more drivers could pass the evaluation agains Euclidean cost constraint, the pruning ratio for Phase II of both methods shows a decreasing trend. Note that, the pruning ratio of *SHAREK*-DP Phase 3-1* also decreases, this is because more and more drivers could satisfy the semi-Euclidean cost constraint when we increase the price.

The final answer is usually less than 4 drivers at the end. The pruning capability goes in reverse with the time consumed for each phase. For example, although Phase III is the most time consuming one, it has, by far, almost the least pruning ratio.

The question that may arise here is: Does it worth to run this phase, even though it does not have high pruning ratio. The answer is definitely yes, it still worth running this phase. For example, consider the case of Figs. 15 and 18, where we have 100,000 drivers, and the final skyline set of drivers returned to the rider is around 4. If we were to apply only the first two phases of SHAREK*, we would prune 97.92% of the drivers in 10 milliseconds. This means that we will end up returning 2,076 unnecessary drivers to the riders in addition to the four that should form the final answer. Instead, should we go ahead with the relatively expensive Phase III, we would return to the rider only the four drivers that form the final answer in 75 milliseconds. With respect to the rider, having four drivers in 75 millisecond is way much better than having 2,080 drivers in 10 milliseconds for three reasons: (1) The rider is likely to be using his cell phone when requesting SHAREK service. Hence, it is of essence to limit the size of the answer to only the right short answer to fit the device small screen. It does worth paying extra 65 milliseconds for this. (2) With 2,076 drivers in hand, it is easy for the rider to make a wrong decision selecting drivers that are more expensive with more waiting time than others. Again, it worth waiting milliseconds to get the right answer and avoid making wrong decisions. (3) When making the request over a web service, either through a cell phone, tablet or even a desktop, the networking cost of downloading the information of 2,076 more drivers may exceed the time we spent in pruning them in Phase III. In fact, the networking cost may dominate the computation cost here. So, spending 65 more milliseconds in computation may end up in saving a lot from the networking cost.

Overall, Figs. 14 to 19 show the need and value for having the three phases working together in achieving the scalability, efficiency, and accuracy of SHAREK*.

6 Related Work

Dynamic ride sharing matching has been an active research area in recent years. Current dynamic ride sharing matching can be classified into four categories: grouping people with similar trip, matching based on historical data, dial-a-ride problem, and finding optimum routes.

The first category is to group multiple ride sharing requests together to achieve the saving goal. Gyozo et al. proposed a trip grouping algorithm [14] to find the ”close by” rider requests based on some heuristics, for example, grouping requests upon expiration time that the trip request must be accommodated. To group the trips where the origin and destination locations of the drivers and riders are close to each other, a fast detour computation method for the driver was proposed [13]. Both methods do not provide waiting time and cost constraints as SHAREK* does, and limited themselves to the similar trips or other heuristics.

The second category of methods perform matching based on historical data, e.g., T-Share [22, 23] and Noah [28]. To find the candidate taxis, T-Share leverages enormous

historical taxi trajectories to predict the future locations of the driver and the query processing is conducted based on this prediction. Thus, the accuracy of its querying results can not be guaranteed in real scenarios. In the mean time, Noah uses a caching scheme to avoid repeated calculation of the same pairs of shortest path and implements a kinetic tree structure that can schedule dynamic requests and adjust routes on-the-fly [16]. Based on this work, San et.al., proposed a series of algorithms to tackle the ride sharing problem by considering the current road conditions [29]. Unlike the pruning techniques of the SHAREK* proposed in our paper, the efficiency of both T-Share and Noah are based on the pre-known of the trips. However, it is too expensive to store every possible shortest path route between any two nodes of the road network in advance. Comparing with using historical data, SHAREK* only uses real time location information of drivers and riders to get the accurate and the best matching for the rider's request, which has less limitations.

The third category is called dial-a-ride problem which refers to the matching between one driver and multiple riders who specify their ride requests, i.e., the time constraints for being picked up and dropped off, between origins and destinations. The main objective of the dial-a-ride problem is to plan a set of m minimum cost driver routes capable of accommodating as many riders as possible, under a set of constraints [10], i.e., travel sales man problem [17], or planing schedules for vehicles with time constraint on each pickup and delivery [3, 9]. Asghari et al. [2] introduce a distributed auction-based framework for taxis where drivers bid on the riders requests, satisfying both the ridern and drivernn constrains. The dial-a-ride problem has been studied in various transport scenarios such as paratransit for handicapped and elderly individuals [4]. There are two aspects that can distinguish our problem from dial-a-ride problem. (1) Contradict to dial-a-ride, each query processing in our problem aims to match multiple drivers against one rider, and (2) The objectives are different as our problem is to find a set of skyline drivers that satisfy the rider time and cost constraints, whereas the dial-a-problem is to plan a set of driver routes.

The last category of ride sharing matching aims to find the optimum routes for drivers and riders. Actually, there exist three types: (1) slugging [26], where the pick-up and drop-off points are pre-assigned by the driver while the rider is required to walk to the meeting point for being picked up and go back to the destination from the drop-off point. It is proved that the computational time complexity of the slugging problem is NP-complete [21]. The matching model of SHAREK* does not belong to this type since no pre-assigned points for pick-up or drop-off are needed. (2) MCR, i.e., mutually beneficial confluent routing [31], where both drivers and riders go from their respective sources to destinations, and they can mutually benefit from traveling together on the confluences of their routes. paper[8] designs an online ride sharing system where passengers have temporal constraints, i.e. earliest departure time and latest arrival time, drivers have distance tolerance. The system will find the qualified drivers if they can send to the passenger his destination on time and can tolerate the detour. SHAREK* is different from this work since no ride sharing cost model is considered in MCR. (3) optimal multi-meeting-point route (OMMPR) query [19], which aims at finding the best route between a source node and a destination node for a given road network such that the weighted average cost between the driver's cost and the total cost of the riders is minimized. The temporal constraints of the maximum waiting time and arrival time are not involved in OMMPR and its solutions, whereas SHAREK* does. Hence, SHAREK* can be distinguished from the matching schemes of this category.

Moreover, there are some spatial search works related to our problem. (1) Online trichromatic pickup and delivery scheduling(OTPD) [32], which aims to find a schedule crowd workers that maximize the trichromatic (worker-item-task) matching utility based on the

demands of item pickup and delivery. The most significant difference between OTPD problem and our problem is that the workers of OTPD only accept tasks within a radius of r while we select all qualified drivers as long as they can satisfy riders constrains. (2) Clue-based route search (CRS) [15], which finds the optimal route according to the user-defined clues. A clue is defined as a triple tuple (w, d, e) where w is a query keyword of POI, d is a user defined distance, and e is a confidence factor. A typical application is to find a travel route according to user demands. However, our problem is different from this work since riders don't have a query keyword and distance constrains. (3) Competitive spatial-temporal searching, which helps mobile agents search for stationary resources on a road network. In this case, the mobile agents are simply assigned to the closest resource, whereas we consider three constrains of riders (the maximum waiting time, the maximum arrival time and the maximum price) when we select drivers. Therefore, these methods can't be applied to our problem neither.

Last but not least, none of the above algorithms in different categories provide the skyline results and take skyline computation as a blessing for efficiency issue, which is also an important character that distinguishes SHAREK* from them.

7 Conclusion

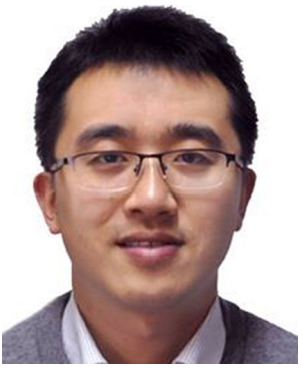
This paper proposes SHAREK*, a scalable and efficient ride sharing system. Drivers who are willing to provide a ride sharing service register themselves with SHAREK* indicating their current locations and destinations. Meanwhile, a rider requesting a ride sharing service would call SHAREK* indicating the rider current location, destination, a maximum price the rider is willing to pay for the service, and a maximum waiting time the rider is willing to wait before being picked up or a maximum arrival time the rider wants to spare for arriving his destination. Then, SHAREK* employs a carefully designed price cost model to find those drivers that can provide the requested ride within the time and price constraints. Among the set of drivers that can provide such service, SHAREK* reports only the skyline set, i.e., maximal vector, of these drivers according to price and waiting time. SHAREK* employs three consecutive phases, namely, *Euclidian Temporal Pruning*, *Euclidean Cost Punning*, and *Semi-Euclidean Skyline-aware pruning*, with the explicit goal to prune as much as drivers as possible *without* the need to calculate actual road network shortest path operations. Extensive experimental evaluation shows that it only takes an average of hundreds of milliseconds from SHAREK* to respond to a ride sharing request with 100,000 drivers around.

Acknowledgment This research was partially supported by following foundations: Zhejiang Provincial Natural Science Foundation of China (LY19F020030).

References

1. Agatz N, Erera A, Savelsbergh M, Wang X (2012) Optimization for dynamic ride-sharing: A review European Journal of Operational Research
2. Asghari M., Deng D., Shahabi C., Demiryurek U., Li Y. (2016) Price-aware real-time ride-sharing at scale: an auction-based approach. In: SIGSPATIAL GIS, p 3
3. Attanasio A, Cordeau JF, Ghiani G, Laporte G (2004) Parallel tabu search heuristics for the dynamic multi-vehicle dial-a-ride problem. Parallel Comput 30(3):377–387

4. Beaudry A, Laporte G, Melo T, Nickel S (2009) Dynamic transportation of patients in hospitals OR Spectrum
5. Börzsönyi S., Kossmann D (2001) Stocker, k.: The Skyline Operator. In: ICDE, pp 421-430. Heidelberg, Germany
6. Cao B, Alarabi L, Mokbel MF, Basalamah A (2015) Sharek: a scalable dynamic ride sharing system. In: MDM
7. Chan EP, Lim H (2007) Optimization and evaluation of shortest path queries. VLDB J. 16(3):343–369
8. Cici B, Markopoulou A, Laoutaris N (2015) Designing an on-line ride-sharing system. In: SIGSPATIAL. ACM, p 60
9. Colorni A, Righini G (2001) Modeling and optimizing dynamic dial-a-ride problems. Int Trans Oper Res 8(2):155–166
10. Cordeau JF, Laporte G (2007) The dial-a-ride problem: models and algorithms. Ann Oper Res 153(1):29–46
11. What do we mean by dynamic ridesharing. http://dynamicridesharing.org/index.php
12. Carpooling-the flinc carpooling service: flinc. https://flinc.org/
13. Geisberger R, Luxen D, Neubauer S, Volker SP, Sanders P, Volker L (2010) Fast detour computation for ride sharing. In: ATMOS, vol 14, pp 88-99. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany
14. Gidofalvi G, Aps G, Risch T, Pedersen TB, Zeitler E (2008) Highly scalable trip grouping for large scale collective transportation systems. In: EDBT, pp 678–689
15. Hu Q, Ming L, Tong C, Zheng B (2019) An effective partitioning approach for competitive spatial-temporal searching (gis cup). In: SIGSPATIAL GIS, pp 620–623
16. Huang Y, Jin R, Bastani F, Wang XS (2015) Large scale real-time ridesharing with service guarantee on road networks. In: PVLDB, pp 2017–2028
17. Kalantari B, Hill AV, Arora SR (1985) An algorithm for the traveling salesman problem with pickup and delivery customers. Eur J Oper Res 22(3):377–386
18. Kung HT, Luccio F (1975) On finding the maxima of a set of vectors. J. ACM 22(4):469–476
19. Li RH, Qin L, Yu JX, Mao R (2016) Optimal multi-meeting-point route search. TKDE 28(3):770–784
20. Lyft On-demand ridesharing. http://www.lyft.me
21. Ma S, Wolfson O (2013) Analysis and evaluation of the slugging form of ridesharing. In: SIGSPATIAL, pp 64–73
22. Ma S, Zheng Y, Wolfson O (2013) T-share: a large-scale dynamic taxi ridesharing service. In: ICDE, pp 410–421
23. Ma S, Zheng Y, Wolfson O (2015) Real-time city-scale taxi ridesharing. TKDE 27(7):1782–1795
24. Nievergelt J, Hinterberger H, Sevcik K (1984) The grid file: an adaptable, symmetric multikey file structure. TODS 9(1):38–71
25. Papadias D, Zhang J, Mamoulis N, Tao Y (2003) Query processing in spatial network databases. In: VLDB, pp 802–813
26. Slugging. http://en.wikipedia.org/wiki/Slugging
27. Thomas Brinkhoff. Network-based generator of moving objects. http://iapg.jade-hs.de/personen/brinkhoff/generator/
28. Tian C, Huang Y, Liu Z, Bastani F, Jin R (2013) Noah: a dynamic ridesharing system. In: SIGMOD, pp 985–988
29. Yeung S, Miller E, Madria S (2016) A flexible real-time ridesharing system considering current road conditions. In: MDM, vol 1. IEEE, pp 186–191
30. Zhang LG, Fang JY, Shen PW (2007) An improved dijkstra algorithm based on pairing heap [j]. J. Image Graphics 5
31. Zhang X, Asano Y, Yoshikawa M (2016) Mutually beneficial confluent routing. TKDE 28(10):2681–2696
32. Zheng B, Huang C, Jensen CS, Chen L, Hung NQV, Liu G, Li G, Zheng K (2020) Online trichromatic pickup and delivery scheduling in spatial crowdsourcing ICDE
33. Zheng B, Su H, Hua W, Zheng K, Zhou X, Li G (2017) Efficient clue-based route search on road networks. TKDE 29(9):1846–1859



Bin Cao received his Ph.D. degree in computer science from Zhejiang University, China in 2013. He then worked as a research associate in Hongkong University of Science and Technology and Noahs Ark Lab, Huawei. He joined Zhe- jiang University of Technology, Hangzhou, China in 2014, and is now an Associate Professor in the College of Computer Science. His research interests include spatio-temporal database and data mining.

Chenyu Hou received his BS in software en- gineering in Zhejiang University of Technology, Hangzhou, China, in 2016. He is now a postgraduate student of the Zhejiang University of Technology. His research interests are spatial database and data mining.

Liwei Zhao received the MS in computer technology from Zhejiang University of Technology in 2017. Currently, he is a Java Development Engineer at Wind Information and Technology, Inc. His research interest is spatial database.

Louai Alarabi received the B.S. from Umm Al-Qura University, Saudi Arabia in computer science, M.S. and Ph.D. degrees in computer science from the University of Minnesota - Twin Cities, MN, USA in 2014 and 2018, respectively. Alarabi is currently an Assistant Professor in the department of computer science at Umm Al-Qura University, Saudi Arabia. His research interests include database systems, spatial data management, big data management, large-scale data analytics, indexing, and main-memory management. His research is published in prestigious research venues, including ACM SIGMOD, ACM SIGSPATIAL, IEEE ICDE, IEEE MDM, and VLDB Journal. His research recognized by the first place and a gold medal award in student research competition at ACM SIGSPATIAL/GIS 2018, among the best paper award at SSTD 2017, Finalist of student research competition at ACM SIGMOD 2017, and best demonstration award at U-Spatial Symposium 2014.

Jing Fan received her B.S., M.S. and Ph.D. degree in Computer Science from Zhejiang Uni- versity, China in 1990, 1993 and 2003. She is now aProfessor of School of Computer Science and Technology at Zhejiang University of Technology, China. She is a Director of China Computer Federation (CCF), and Chairman of Chapter Hangzhou of CCF. Her current research interest includes middleware, virtual reality and visualization.

Mohamed F. Mokbel (Ph.D., Purdue University, MS, B.Sc., Alexandria University) is a Professor in the Department of Computer Science and Engineering, University of Minnesota. His research interests include the interaction of GIS and location-based services with database systems and cloud computing. His research work has been recognized by the VLDB 10- Years Best Paper Award, five Best Paper Awards, and by the NSF CAREER award. Mohamed has held prior visiting positions at Microsoft Research and Hong Kong Polytechnic U., and is a co-founder of the GIS Technology Innovation center in Saudi Arabia. Mohamed is/was the program co-chair for ACM SIGMOD 2018, ACM SIGSPATIAL GIS from 2008 to 2010, and IEEE MDM 2011 and 2014. He is Editor-in-Chief for Springer Distributed and Parallel Databases journal, and Associate Editor for ACM Books, ACM TODS, ACM TSAS, VLDB journal, and GeoInformatica. Mohamed is an elected Chair of ACM SIGSPATIAL 2014-2017.

Anas Basalamah is an Associate Professor at the Computer Engineering Department of Umm Al Qura University. He is a cofounder and director of the Wadi Makkah Technology Innovation Center at Umm Al-Qura University. He did his MSc and PhD Degrees at Waseda University, Tokyo in 2006, 2009 respectively. He worked as a Post Doctoral Researcher at the University of Tokyo and the University of Minnestoa in 2010, 2011 respectively. He cofounded Averos and Hazen.ai. His areas of interest include; Embedded Networked Sensing, Smart Cities, Ubiquitous Computing, Participatory and Urban Sensing.