SHAREK: A Scalable Dynamic Ride Sharing System

Bin Cao*, Louai Alarabi†, Mohamed F. Mokbel†, Anas Basalamah‡
*College of Computer Science and Software Engineering, Zhejiang University of Technology, Hangzhou, China
Email: bincao@zjut.edu.cn
†Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA
Email: {louai,mokbel}@cs.umn.edu
‡Computer Engineering Department and KACST GIS Technology Innovation Center, Umm Al-Qura University
Makkah, Saudi Arabia
Email: ambasalamah@uqu.edu.sa

***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 the maximum price they are willing to pay for the service and the maximum waiting time before being picked up. 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 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.**

I. 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 [1]. Since dynamic ride sharing can be enabled by smart 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 [2], [3]. As a result, a number of dynamic ride sharing systems are available nowadays, e.g., Flinc [4], Lyft [5], Noah [6]. 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. Meanwhile, 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. 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, 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 few milliseconds to satisfy the rider request, even if there are 10,000 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) We provide experimental evidence of the scalability and efficiency of our proposed algorithm.

The rest of this paper is organized as follows. Section II sets the stage for various concepts used in SHAREK. Section III discusses SHAREK query processing. Experimental evaluation is presented in Section IV. Section V highlights related work. Finally, Section VI concludes the paper.

II. 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.

A. 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 $dest$. With such registration, SHAREK is allowed to track their locations to asses their suitability for any ride sharing service. Once the driver reaches to his destination, the driver is unregistered from SHAREK. 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_Time) that r can afford before being picked up, and (4) 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 waiting time and price constraints.

B. 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 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 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 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 [7] (also known as the *skyline* query in database context [8]) to return only the set of the two dimensioanl *skyline* drivers in terms of waiting 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 and less price than that of d_i. Meanwhile, if d_j has less waiting time and less price

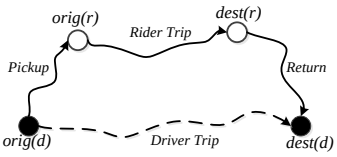


Fig. 1. The illustration for different costs

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.

C. Price Cost Model

Figure 1 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 her 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 Figure 1, 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 Figure 1) and its original route (the dotted line in Figure 1). 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 $orig(r)$.

D. 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_Time} 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) $Pickup(d, r) < r_{max\_Time}$, (2) $Price(d, r) < r_{max\_Price}$, and (3) d is in the set of skyline drivers based on $Pickup(d, r)$ and $Price(d, r)$.

It is important to note here that in the first condition we compare $Pickup(d, r)$, which is a a shortest path distance with r_{max_Time}, 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.

E. 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 for updating the value of *DriverTrip*.

Grid Index. The *Driver Table* is indexed by a simple gird index [9] 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.

III. SHAREK Query Processing

A naive way to support the ride sharing query 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

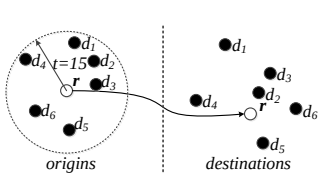
satisfy both the pickup time and cost constraints, run a two-dimensional skyline algorithm over pickup time and cost to get those drivers that are not dominated by any other drivers. Though the solution looks simple, it has a prohibitive cost of computing large numbers of shortest paths, which is not 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 Figure 2(a) throughout the whole section. 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. We assume that maximum waiting time and maximum price rider constrains are 15 and 30, respectively.

A. 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, 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. For this phase, this means two important issues: (1) There is no shortest path computations in this phase at all, hence, we can early prune a set of drivers without expensive computations, and (2) Not all the output candidate drivers can make it to the rider within the requested time, as the actual road network distance from the driver to the rider is more than the computed Euclidean distance. Hence, a driver d that satisfies the temporal constraint based on the Euclidean distance may not actually satisfy the temporal constraint based on the road network distance.

The main idea of this step 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 maximum waiting time constraint of the rider r. 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 main reason behind this is that the Euclidean distance between d and r does not allow d to be at r location in time. Since the road network distance is guaranteed to be more than the Euclidian distance, then driver d cannot make it to the rider r using a road network distance. The set



(a) Range query

ID	Pickup	Euclidean Pickup	Return	Euclidean Return	Driver Trip
d_1					8
d_2					9.2
d_3					10
d_4					9.6
d_5					10
d_6					13.4

(b) *Matching Table*

Fig. 2. Temporal pruning with r_{max_Time}

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 2(a) gives the temporal pruning result for our running example. A range query is submitted for retrieving drivers that are within Euclidean distance of the rider's maximum waiting time $r_{max\_Time} = 15$. As a result, six drivers (i.e., $d_1, d_2, ..., d_6$) are selected.

B. 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.

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:

$$EuclideanPrice(d, r) = EuclideanPickup(d, r) + 2 * RiderTrip(r) + EuclideanReturn(d, r) - DriverTrip(d) \quad (2)$$

Comparing Equations 1 and 2, 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. Algorithm 1 gives the pseudo code for the first two phases. First, we obtain the rider r information, i.e., its location and temporal constraint. Then, 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* (Figure 2(b)), 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

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*

```
1 origin(r) ← get the origin of r;
2 r_max_Time ← get the maximum waiting time of r;
3 Matching Table ← Q_R (origin(r), r_max_Time, g_index)
  RiderTrip ← compute the shortest path cost between orig(r) and
  dest(r);
4 for d ∈ Matching Table do
5     EuclideanPickup(d, r) ← compute the Euclidean distance
      between orig(d) and orig(r) ;
6     EuclideanReturn(d, r) ← compute the Euclidean distance
      between dest(r) and dest(d) ;
7     if EuclideanPickup(d, r) + 2 * RiderTrip(r) +
      EuclideanReturn(d, r) − DriverTrip(d) > r_max_Price then
      // Equation 2
8         remove d from Matching Table;
9 return A set of candidate drivers stored in Matching Table
```

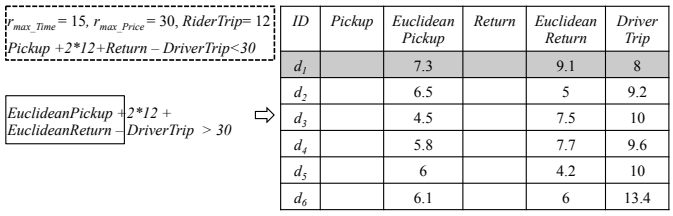


Fig. 3. Cost pruning with r_{max_Price}

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 (Equation 2) 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 computations for d.

Example. Figure 3 gives the *Matching Table* after we compute the Euclidean Pickup and Euclidean Return costs for each of the six drivers d_1 to d_6 that are produced from Phase I, assuming that the temporal constraint $r_{max\_Time} = 15$. Assume that the shortest path cost for $RiderTrip(r) = 12$, then we calculate Equation 2 for each of the six drivers. We find that the total cost for driver d_1 is actually more than 30, which is the maximum price set by the rider r. Hence, we decide to remove driver d_1 from the matching table without any shortest path computation, as we are sure that d_1 will never make it to the final result as its road network distance will exceed its Euclidean distance. The rest of drivers d_2 and d_6 are still candidates as their total cost (using Euclidean distance) is less than 30, hence they still have a chance to make it.

C. 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.

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 Equation 2. In particular, we use the following equation:

$$\begin{aligned} SemiEuclideanCost(d, r) = \\ Pickup(d, r) + 2 * RiderTrip(r) + \\ EuclideanReturn(d, r) - DriverTrip(d) \end{aligned} \quad (3)$$

Comparing Equations 3 and 2, 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 [10] 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 make it on time, 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 make it on time . 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 (Equation 3) 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 (Equation 3) 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 (Equation 3) is less than the MAX value, and its computed actual total cost (Equation 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.

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

1 calculate $(EuclideanReturn(d, r) - DriverTrip(d))$ for each driver d in *Matching Table*;
2 sort *Matching Table* by values of $(EuclideanReturn(d, r) - DriverTrip(d))$ in an ascending order;
3 $MAX \leftarrow r_{max\_Price}$;
4 **while** *Matching Table* $\neq \emptyset$ **do**
5 $Pickup(d, r) \leftarrow$ retrieve the nearest driver d in *Matching Table* from $orig(r)$; `// INN query`
6 **if** $Pickup(d, r) > r_{max\_Time}$ **then**
7 break ; `// terminate the program`
8 **if** $Pickup(d, r) + 2 * RiderTrip(r) + EuclideanReturn(d, r) - DriverTrip(d) > MAX$ **then** `// Equation 3`
9 remove d and drivers that below d from *Matching Table*;
10 **else**
11 $Return \leftarrow$ compute the shortest path cost between $dest(r)$ and $dest(d)$;
12 **if** $Pickup(d, r) + 2 * RiderTrip(r) + Return(d, r) - DriverTrip(d) > MAX$ **then** `// Equation 1`
13 remove d from *Matching Table*;
14 **else**
15 $MAX \leftarrow Pickup(d, r) + 2 * RiderTrip(r) + Return(d, r) - DriverTrip(d)$;
16 $R \leftarrow$ move d from *Matching Table* to the result set;
17 **return** skyline driver set R

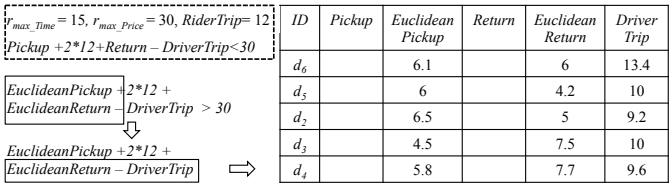


Fig. 4. *Matching Table* sorting

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

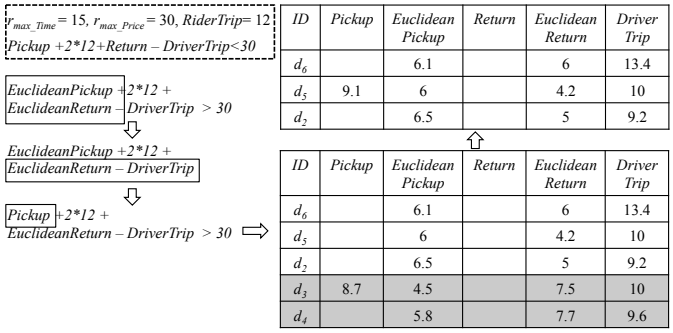


Fig. 5. Filtering based on Equation 3

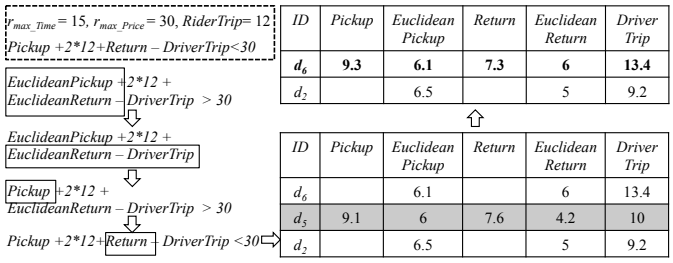


Fig. 6. Filtering based on Cost Constraint

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 (Equation 3). 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 4 gives the *Matching Table* sorted based on $(EuclideanReturn - DriverTrip)$ in an ascending order. Then, *Semi-Euclidean Distance Pruning* will iterate over drivers by querying incremental nearest neighbor. Figure 5 gives the procedure for filtering based on the estimation described in Equation 3. First, INN query retrieves the nearest driver d_3. We find that the cost of d_3 is more than the rider constraint $(8.7 + 2 * 12 + 7.5 - 10 > 30)$, hence driver d_3 is removed from *Matching Table*. In addition, based on the conclusion of *Matching Table* sorting, driver d_4 is also filtered out without any shortest path computation.

Figure 5 also gives the next iteration, where we can see that after filtering out d_3 and d_4 (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_5 is chosen, where its $Pickup$ cost is not

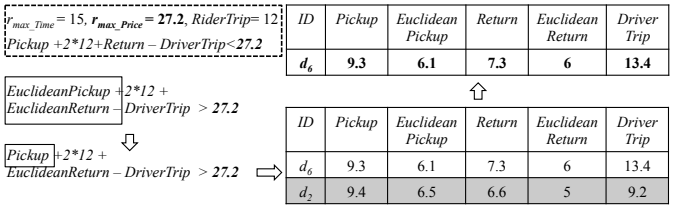


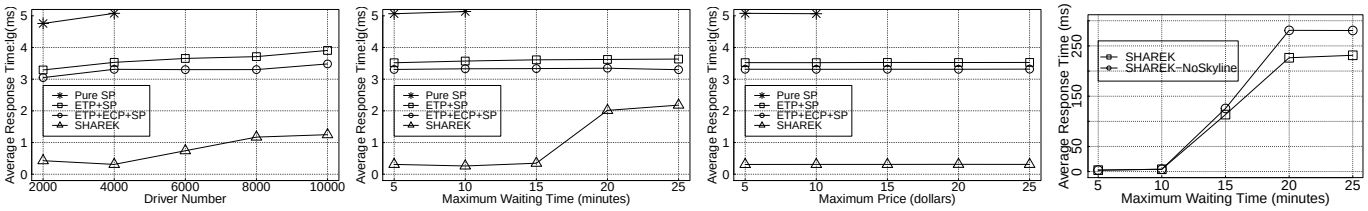
Fig. 7. Skyline processing

only within the 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_5. As shown in Figure 6, we use the ride sharing cost constraint (Equation 1) shown in the left bottom of the figure to test d_5. Unfortunately, d_5 can not meet the demand of the rider's maximum price, i.e., d_5 provide a price of 30.7 $(9.1+2*12+7.6-10)$, which is greater than 30. As a result, d_5, denoted by the grey area in lower right table, is removed from *Matching Table*. Then, searching the nearest driver in next iteration. This time, d_6 is chosen where its $Pickup$ and $Return$ costs can satisfy all the relations listed in the left part of the figure. Since d_6 passes the last evaluation of cost constraint (Equation 1), driver d_6 becomes a qualified one.

Figure 7 describes the skyline processing case. The driver d_6 is the first found qualified driver with ride sharing cost 27.2, which is less than current MAX (30), and the time for picking up is 9.3 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 27.2. Thus, to find this driver in the next iteration, we update the maximum price/ride sharing cost r_{max_Price} to 27.2 as shown in the top left part of the figure. Based on this new value, we filter out driver d_2 when evaluating Equation 3 $(9.4 + 2 * 12 + 5 - 9.2 > 27.2)$. Eventually, only one driver d_6 out of 10 is returned to the rider r.

IV. Experimental Evaluation

This section provides experimental evaluation of SHAREK based on an actual system implementation. We first compare the overall performance of SHAREK (Section IV-A), then we investigate the performance of each phase in SHAREK (Section IV-B). 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 San Francisco, CA, USA, containing 223,606 edges and 175,343 nodes. The synthetic data set are the drivers and riders on the road network, which are generated according to Brinkhoff road network generator [11].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 40 km per hour, the ride sharing cost is one dollar per KM. Drivers are indexed by a 32×32 grid index. We implement a relatively efficient shortest path search algorithm, namely, Bidirectional Dijkstra [12], which searches the shortest path from two sides simultane-



(a) Varying Driver Number (b) Varying Maximum Waiting Time (c) Varying Maximum Price (d) Skyline VS No Skyline

Fig. 8. Comparison study for SHAREK

ously and achieves better performance than classical Dijkstra algorithm in practice [13]. All experiments are evaluated on a server machine with Intel(R) Xeon(R) CPU E5-2637 3.50 GHz processor and 8 GB RAM with Ubuntu Linux 14.04.

A. Overall Performance

This section studies the average response time of SHAREK. In Figure 8, we compare SHAREK with the following three alternatives: (1) *Pure-SP*, where we do an exhaustive search by computing the shortest path cost for all drivers (using bidirectional Dijkstra algorithm), (2) *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, (3) *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 order to avoid skewness towards large values of long matching time, all experiments in Figure 8 are plotted with a logarithmic scale of base 10.

In Figure 8(a), we vary the number of drivers from 2,000 to 10,000, while fixing the rider waiting time constraints to 5 minutes and the maximum price to 5 dollars. The *Pure-SP* method is clearly the most inefficient method (four orders of magnitude slower than SHAREK). Due to its inefficiency, we could only plot its values for up to 4,000 drivers. Such inefficiency is basically due to the many time-consuming shortest path search operations. In the mean time, 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. 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 Figure 8(b), we vary the maximum waiting time from 5 to 25 minutes, while fixing the number of drivers to 4000 and the maximum price to 5 dollars. Performance comparison among the four techniques follow the same trends as that of Figure 8(a). The *Pure-SP* method is not practical, where we cannot run it for more than 10 minutes waiting time. The *ETP+SP* and *ETP+ECP+SP* methods both show a stable performance that is not affected by the increase in the waiting time. This is because the maximum price here results in more pruning than the waiting time. Hence, no mater how much we increase the waiting time, we will still end up with the same number of drivers pruned by the maximum cost, and hence the performance is stable. SHAREK has performance that is up to two orders of magnitudes better than that of *ETP+SP* and *ETP+ECP+SP*. This shows the pruning capability of SHAREK. Unlike the cases for *ETP+SP* and *ETP+ECP+SP*, the performance for SHAREK 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.

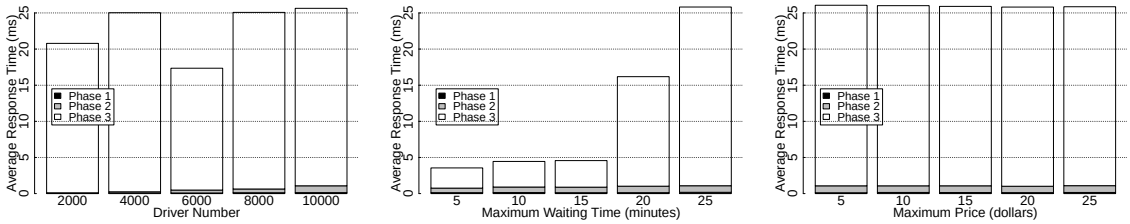
In Figure 8(c), we vary the maximum price from 5 to 25 dollars, while fixing the number of drivers to 4,000 and the waiting time to 5 minutes. The difference in performance among various algorithms remains the same as in Figures 8(a) and 8(b). However, all methods here show a stable performance with the increase in the maximum price constraints. This shows that the temporal pruning with a small cost pruning in this set of drivers is enough to prune the most of the drivers we want. Hence, the performance becomes stable here, which is in favor to SHAREK and its variants.

Figure 8(d) compares SHAREK against another alternative *SHAREK-No Skyline*, where we use all the pruning techniques we have, except skyline pruning. The driver number is fixed to 10,000. It is interesting to see that *SHAREK-No Skyline* and SHAREK almost share the same performance when maximum waiting time is within 10 minutes. The performance of SHAREK is getting better and better as the maximum waiting time increases. This shows that adding the skyline functionality to the user constraints does not result in any extra overhead. Instead, SHAREK takes advantage of the skyline functionality to increase the performance as well as reporting more meaningful answer, i.e., less number of drivers.

B. Inside SHAREK

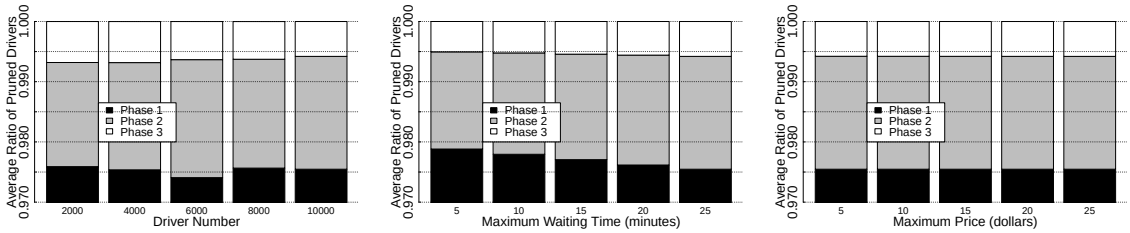
This section studies the internals of SHAREK in terms of the performance and pruning power of each phase separately.

1) Response Time for Each Phase: Figure 9 gives the breakout of response time for the three phases of SHAREK, when increasing number of drivers, maximum waiting time, and maximum price. The maximum waiting time, maximum price, and number of drivers is set to 25 minutes, 25 dollars and 10,000 drivers. It is clear that 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. This is expected as the first two phases do not encounter any shortest path computations. Meanwhile, the



(a) Varying Driver Number (b) Varying Maximum Waiting Time (c) Varying Maximum Price

Fig. 9. Average Response Time for Different Phases of SHAREK



(a) Varying Driver Number (b) Varying Maximum Waiting Time (c) Varying Maximum Price

Fig. 10. Average Ratio of Pruned Drivers for Different Phases of SHAREK

third phase is the only one that needs to make actual shortest path computations in addition to the skyline pruning.

In Figure 9(a), the fluctuation of average response time when increasing the number of drivers could be explained by the reason that the distribution of different sets of drivers in the road network is different. Hence, the number of qualified drivers that satisfy the rider's constraints does not necessarily increase, which can cause the decrease trend when the driver number increases after 4,000 as shown in Figure 9(a). Similar case also occurred in Figure 8(a), where a slight decrease is after 2,000 driver number.

Figure 9(b) shows a similar trend as that of Figure 8(b), the increase of maximum waiting time can finally cause significant influences on the response time of SHAREK. On the contrary, combining Figures 9(c) and 8(c) shows that the average response time is less sensitive to the increase of the maximum price. The main reason behind this is that maximum price can use embedded skyline operation to offset the overhead from the increase of its value.

2) Pruning Ratio for Each Phase: Figure 10 gives the pruning capability of each phase separately. The settings of the experiments are the same as of Figure 9. Phase I clearly has the most pruning capability with more than 97.5%, followed by Phase II with around 2%, then Phase III with around 0.5%. The final answer is usually less than 5 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, 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 Figures 9 and 10, where we have 10,000 drivers, and the final skyline set of drivers returned to the rider is around 5. If we were to apply only the first two phases of SHAREK, we would prune 99.5% of the drivers in two milliseconds. This means that we will end up returning 45 unnecessary drivers to the riders in addition to the five 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 five drivers that form the final answer in 25 milliseconds. With respect to the rider, having five drivers in 25 millisecond is way much better than having 50 drivers in 2 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. (2) With 50 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. (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 50 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.

Overall, Figures 9 and 10 shows the need and value for having the three phases working together in achieving the scalability, efficiency, and accuracy of SHAREK.

V. Related Work

Current dynamic ride sharing matching can be classified into four categories. 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 [15]. 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 [3] and Noah [6]. 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. 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]. Both T-Share and Noah depend on the pre-known of the trips. However, SHAREK only uses real time location information and has less limitation, and it can be viewed as an addition to them to enhance their efficiency when the historical data is not available.

The third category of ride sharing matching is called slugging [17], in which 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 [18]. The matching model of SHAREK does not belong to this category since no pre-assigned points for pick-up or drop-off are needed.

The last category is called dial-a-ride problem which refers to the matching between one driver and multiple riders who specify their ride requests. 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 [19], i.e., travel sales man problem [20], or planing schedules for vehicles with time constraint on each pickup and delivery [21]. 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.

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.

VI. 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. 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 few milliseconds from SHAREK to respond to a ride sharing request with 10,000 drivers around.

Acknowledgment

This work was supported by the KACST National Science and Technology Plan under grant #11-INF2061-10, and the KACST GIS Technology Innovation Center at Umm Al-Qura University under grant #GISTIC-14-02. In addition, This work is partially supported by the National Science Foundation, USA, under Grants IIS-0952977 and IIS-1218168.

References

[1] "What do we mean by dynamic ridesharing," http://dynamicridesharing.org/index.php.
[2] N. Agatz, A. Erera, M. Savelsbergh, and X. Wang, "Optimization for dynamic ride-sharing: A review," *European Journal of Operational Research*, 2012.
[3] S. Ma, Y. Zheng, and O. Wolfson, "T-share: A large-scale dynamic taxi ridesharing service," in *ICDE*, 2013, pp. 410–421.
[4] "Carpooling-the flinc carpooling service: flinc," https://flinc.org/.
[5] "Lyft: On-demand ridesharing," http://www.lyft.me.
[6] C. Tian, Y. Huang, Z. Liu, F. Bastani, and R. Jin, "Noah: A dynamic ridesharing system," in *SIGMOD*, 2013, pp. 985–988.
[7] H. T. Kung, F. Luccio, and F. P. Preparata, "On Finding the Maxima of a Set of Vectors," *Journal of ACM*, vol. 22, no. 4, pp. 469–476, 1975.
[8] S. Börzsönyi, D. Kossmann, and K. Stocker, "The Skyline Operator," in *ICDE*, Heidelberg, Germany, Apr. 2001, pp. 421–430.
[9] J. Nievergelt, H. Hinterberger, and K. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *TODS*, vol. 9, no. 1, pp. 38–71, 1984.
[10] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *VLDB*, 2003, pp. 802–813.
[11] "Thomas brinkhoff: Network-based generator of moving objects," http://iapg.jade-hs.de/personen/brinkhoff/generator/.
[12] I. Pohl, *Bi-directional search*. IBM TJ Watson Research Center, 1970.
[13] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou, "Shortest path and distance queries on road networks: an experimental evaluation," *PVLDB*, vol. 5, no. 5, pp. 406–417, 2012.
[14] G. Gidofalvi, G. Aps, T. Risch, T. B. Pedersen, and E. Zeitler, "Highly scalable trip grouping for large scale collective transportation systems," in *EDBT*, 2008, pp. 678–689.
[15] R. Geisberger, D. Luxen, S. Neubauer, P. S, and L. Volker, "Fast detour computation for ride sharing," in *ATMOS*, vol. 14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010, pp. 88–99.
[16] Y. Huang, R. Jin, F. Bastani, and X. S. Wang, "Large scale real-time ridesharing with service guarantee on road networks," in *PVLDB*, 2015, pp. 2017–2028.
[17] "Slugging," http://en.wikipedia.org/wiki/Slugging.
[18] S. Ma and O. Wolfson, "Analysis and evaluation of the slugging form of ridesharing," in *SIGSPATIAL*, 2013, pp. 64–73.
[19] J.-F. Cordeau and G. Laporte, "The dial-a-ride problem: models and algorithms," *Annals of Operations Research*, vol. 153, no. 1, pp. 29–46, 2007.
[20] B. Kalantari, A. V. Hill, and S. R. Arora, "An algorithm for the traveling salesman problem with pickup and delivery customers," *European Journal of Operational Research*, vol. 22, no. 3, pp. 377–386, 1985.
[21] A. Attanasio, J.-F. Cordeau, G. Ghiani, and G. Laporte, "Parallel tabu search heuristics for the dynamic multi-vehicle dial-a-ride problem," *Parallel Computing*, vol. 30, no. 3, pp. 377–387, 2004.