Large Scale Real-time Ridesharing with Service Guarantee on Road Networks [*]

Yan Huang
University of North Texas
huangyan@unt.edu

Favyen Bastani
Massachusetts Institute of Technology
fbastani@mit.edu

Ruoming Jin
Computer Science
Kent State University
jin@cs.kent.edu

Xiaoyang Sean Wang
School of Computer Science
Shanghai Key Laboratory of Data Science
Fudan University
xywangcs@fudan.edu.cn

ABSTRACT

Urban traffic gridlock is a familiar scene. At the same time, the mean occupancy rate of personal vehicle trips in the United States is only 1.6 persons per vehicle mile. Ridesharing has the potential to solve many environmental, congestion, pollution, and energy problems. In this paper, we introduce the problem of large scale real-time ridesharing with service guarantee on road networks. Trip requests are dynamically matched to vehicles while trip waiting and service time constraints are satisfied. We first propose two scheduling algorithms: a branch-and-bound algorithm and an integer programing algorithm. However, these algorithms do not adapt well to the dynamic nature of the ridesharing problem. Thus, we propose kinetic tree algorithms which are better suited to efficient scheduling of dynamic requests and adjust routes on-the-fly. We perform experiments on a large Shanghai taxi dataset. Results show that the kinetic tree algorithms outperform other algorithms significantly.

1. INTRODUCTION

Urban and metropolitan areas are growing at tremendous rates and already host more than half of the entire human population. In an urban city like Shanghai, there are approximately 120,000 road intersections, 40,000 taxis, and more than 400,000 taxi trips per day (these numbers are derived from our experimental dataset). Slight changes in weather such as light rain will send the city into a gridlock. Despite mounting energy, pollution, and congestion problems, many vehicles continue to travel with empty seats. The mean occupancy rate of personal vehicle trips in the United States is only 1.6 persons per vehicle mile [10]. In 1999, if 4% of drivers had rideshared, it would have offset the increase in congestion in the 68 examined urban areas completely [8]. Large scale private car or taxi sharing is becoming increasingly popular. Tickengo [23], founded in 2011, is an open ride system where over 50,000 people participate in ridesharing. Uber and Lyft allow for peer-to-peer ride matching through mobile-phone applications. Other companies include Didi, Kuaidi, Avego, PickupPal, Zimride, and Zebigo. For most ridesharing systems today, the main operation modes are: (1) a driver gives/shares ride to/with a passenger; (2) a small set of trips with same origin or destination, e.g. airport, are pre-arranged. A system where multiple passengers/trips can be combined in real-time with service guarantees has many challenges and has not been realized; this is the focus of our paper.

Real-time ridesharing [23, 11], enabled by low cost geo-locating devices, smartphones, wireless networks, and social networks, is a service that dynamically arranges ad-hoc shared rides. In a real-time ridesharing with service guarantees on road networks problem (hereafter referred to simply as ridesharing), a set of servers travel over a road network, cruising when not committed to any service and delivering passengers otherwise. Requests for rides are received over time, each consisting of two points, a *source* and a *destination*. Each request also specifies two constraints, a *waiting time*, defining the maximal time allowed between making the request and receiving the service, and a *service constraint*, defining the acceptable extra detour time from the shortest possible trip duration. When a new request is received, it is evaluated immediately for server matching and scheduling. In order to be assigned to the request, a server must satisfy all constraints, both those of the new request and those of requests already assigned to the server. *The goal is to schedule requests in real-time and minimize the servers' travel times to complete all committed services while meeting service guarantees.*

However, providing ridesharing service at the urban scale is a non-trivial problem. The core issue is to devise a real-time matching algorithm that can quickly determine the best vehicle (taxi, cab, bus) to satisfy incoming service requests. The traditional dial-a-ride problem [9] aims to design vehicle routes and schedules for small to medium sized trip and vehicle sets, e.g. a few vehicles serving tens of requests, focusing on scenarios where requests are known ahead of time and servers originate and finish at known depots. These approaches are not designed to deal with the enormity of modern situations. Additionally, the dynamic and en route nature renders many of these algorithms either inapplicable or inefficient.

[*]Work by Huang was partially supported by the National Science Foundation Under Grant No. IIS-1017926. Work by Jin is partially supported by NSF CAREER grant IIS-0953950, NIH/NIGMS grant: R01GM103309, and OSC (Ohio Supercomputer Center) grant: PGS0218. Work by Wang is partially supported by NSFC grant #61370080. We would like to thank anonymous reviewers for their valuable comments.

Proceedings of the VLDB Endowment, Vol. 7, No. 14
Copyright 2014 VLDB Endowment 2150-8097/14/10.

In this paper, we focus on developing fast matching algorithms for large scale real-time ridesharing. Our algorithms are applicable to existing services including taxi services, private vehicle sharing, elevator systems, minibus services, and courier services. We have a demo of our ridesharing system [22] (a demo is available at: http://hpproliant.cse.unt.edu/noah/) and this paper describes the core algorithms of the system.

We acknowledge that there are other important factors which need to be considered for large scale real-time ridesharing, such as inter-personal, safety, social discomfort, and pricing concerns. Possible solutions include real-name profiling, reputation, or social network trust building systems [10]. However, those are beyond the scope of this paper.

1.1 Problem Definition

A road network $G = \langle V, E, W \rangle$ consists of a vertex set V and an edge set E. Each edge $(u, v) \in E$ $(u, v \in V)$ is associated with a weight $W(u, v)$ indicating the traveling cost along the edge (u, v); this traveling cost may be a time or distance measure. *Assuming driving speeds are available, time and distance can typically be converted from one to the other, and here they are used interchangeably.*

Given two nodes s and e in the road network, a path p between them is a vertex sequence $(v_0, v_1, \cdots, v_k)$, where (v_i, v_{i+1}) is an edge in E, $v_0 = s$, and $v_k = e$. The path cost $W(p) = \sum W(v_i, v_{i+1})$ is the sum of all edge costs $W(v_i, v_{i+1})$ along the path. The shortest path cost $d(s, e)$ is defined as the minimal cost for paths linking from s to e, i.e., $d(s, e) = \min_p W(p)$; the corresponding path with cost $d(s, e)$ is a shortest path from s to e.

DEFINITION 1. *(**Trip Request**) A trip request $tr = \langle s, e, w, \epsilon \rangle$ with respect to a road network $G = \langle V, E, W \rangle$ is defined by a source $s \in V$, a destination $e \in V$, a maximal waiting time w (the maximal time allowed between making the request and receiving the service), and a service constraint ϵ (the extra detour acceptable in a trip, bounding the overall time from s to e by $(1 + \epsilon)d(s, e)$).*

We consider a unified waiting time w and service constraint ϵ for all requests specified by the service provider. However, our proposed algorithms can be easily generalized to request-specific constraints. We further assume that G is static over time (e.g., we do not consider different path costs at different times of the day), but the algorithms we present can handle the case where G changes under a predetermined pattern.

An accepted trip request tr has an assigned server (e.g., a taxi or private vehicle). The server should pick up the rider at s and drop off the rider at e while satisfying the constraints; the request is completed after the rider is dropped off.

To deal with real-time ride sharing, for each trip $tr_i = \langle s_i, e_i, w, \epsilon \rangle$ and its assigned server, we further introduce r_i, the server's location when the request is made; and for each server, the server's trip set $TR = \{tr_1, tr_2, \ldots, tr_m\}$ consists of all trip requests assigned to the server (both completed and uncompleted). Given this, a general *trip schedule S* for a server with trip set of size m can be described in a sequence of $3m$ elements, $(x_1, x_2, \cdots, x_{3m})$, where an element x_j in the sequence is either a trip source point (s_i, where a rider is picked up), a trip destination point (e_i, where a rider is dropped off), or a trip request point (r_i, where a request is received). Furthermore, a server is assumed to travel along the shortest path in the road network when moving between two consecutive points in the trip schedule x_i and x_{i+1}. Thus, the *trip cost* $d_T(x_i, x_j)$ between any two points (x_i, x_j) in the trip schedule is denoted as

$$d_T(x_i, x_j) = d(x_i, x_{i+1}) + d(x_{i+1,i+2}) + \cdots + d(x_{j-1}, x_j).$$

The overall trip cost is simply $d_T(x_1, x_{3m})$.

Figure 1 illustrates a trip schedule for four trip requests where the four pickups happen to be before any dropoff. Note that each moving server is associated with a trip schedule at any given time. A trip request tr_i is active at time t if the request has been accepted but not yet completed. Then, each server is associated with a subset of *active trips*. For instance, in Figure 1, the active trips are $\{tr_1, tr_2, tr_3\}$ at time t_1; $\{tr_1, tr_2, tr_3, tr_4\}$ at time t_2; and $\{tr_1, tr_2, tr_4\}$ at time t_3.

Some trip schedules do not meet the service guarantees for each trip request in the schedule. We thus formally introduce the concept of a *valid trip schedule*.

DEFINITION 2. *(**Valid Trip Schedule**) A valid trip schedule $S = (x_1, x_2, \cdots, x_{3m})$ for a trip set TR satisfies three conditions:*

1. **Point order** *For any trip tr_i, let $x_{i_1} = r_i$, $x_{i_2} = s_i$, and $x_{i_3} = e_i$. Then, we must have $i_1 < i_2 < i_3$ (index represents the position of the points in the trip schedule S), i.e., the requesting point must happen before the pickup point, which must happen before its destination point;*

2. **Waiting time constraint** *For any trip tr_i, the distance (waiting time) from the server's location when the request is made to the request's pickup point should be smaller than the waiting time constraint, i.e., $d_T(r_i, s_i) \le w$;*

3. **Service constraint** *For any trip tr_i, the actual travel distance from the pickup point to the dropoff point $d_T(s_i, e_i)$ should not be more than the shortest distance between them multipled by the service constraint, i.e., $d_T(s_i, e_i) \le (1 + \epsilon)d(s_i, e_i)$.*

To formally define the *real-time ridesharing* problem, we further introduce the *augmented valid trip schedule*: *Assuming at time t, there are m active trips for a given server, let the current valid trip schedule be $(x_1, x_2, \cdots, x_{3m})$, where t is between x_c and x_{c+1}. For a new trip request tr_{m+1} at time t, the augmented valid trip schedule is $(x'_1, x'_2, \cdots, x'_{3m+3})$, where $x'_i = x_i$ for $i \le c$, and $x'_{c+1} = r_{m+1}$.* In other words, the augmented valid trip schedule combines a new request with existing requests and shares the same partial trip schedule before the new request is made at time point t. Also any augmented valid trip schedule consists of two parts: the *finished schedule* $(x_1, x_2, \cdots, x_c, r_{m+1})$ and the new *unfinished schedule* $(x'_{c+2}, \cdots, x'_{3m+3})$.

The problem of **Real-Time Ridesharing** *is: Given a set of vehicles on the road network G and a new incoming request tr, find the vehicle that minimizes the overall trip cost for the augmented valid trip schedule.*

The *scheduling capacity* c is a limit on the number of active trips that can be scheduled to a vehicle. The *scheduling workload per vehicle* is the number of active trips that needs to be scheduled by the algorithm for a vehicle.

Note that since the finished schedule cannot be changed (because it has already been executed), we essentially need to find the minimum trip cost for the unfinished schedule. We also observe that the minimum cost is useful to determine the best match between an incoming trip request and the available vehicles in a real-time fashion. The minimum cost, then, is greedy in nature: When additional new requests come in, the past optimal matching between a trip request and the server may not be the minimum anymore. However, in real-time, this type of optimality tends to be the best we can achieve and can be easily understood and accepted by riders as the future requests are not available . We choose not to batch process

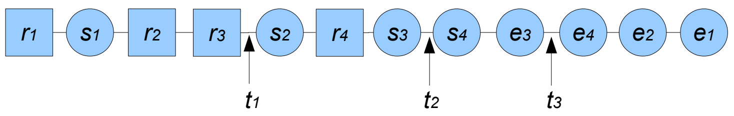

Figure 1: Trip Schedule. s_i**: trip starting point;** e_i**: trip ending point;** r_i **server location when request of trip** tr_i **comes in.**

requests at a fixed time interval and prefer instant feedback to the users. The batch processing may achieve better system scheduling but is based on sacrificed user experience, e.g. standing by curbside and waiting for 5 minutes to know the results. Furthermore, our system is user-centered and does not try to violate a user's service constraints in order to improve overall system performance.

Finally, we note that the problem of real-time ridesharing is NP-hard as the classical Hamiltonian path problem can be reduced to this problem (assuming all the trips have the same ending points and requested in almost the same time). For simplicity, the details of NP-hardness proof is omitted here.

1.2 Challenges

The main challenge in ridesharing is to determine how to handle trip requests as they flow into the system in real-time. From a server's point of view, for any new request, the server may have already selected (and be executing) a trip schedule for its existing customers. Given this, how can we quickly help it to determine whether it can accommodate a new request? Note that in order to respond to such a request, one may have to reshuffle the predefined schedule and the reshuffled one has to be a valid schedule.

Furthermore, in a large metropolitan area such as Shanghai, the number of requests can be very large, especially during rush hours. Clearly, for a trip request tr_i, servers that are farther than w from the pickup location are unable to respond to the request. Even though potential servers can be filtered through a dynamic spatial indexing structure [21, 18, 15] on the moving servers, the existing approaches can still be very computationally expensive and result in low response times.

Most current algorithms are designed for offline computation. The approaches that use branch-and-bound [16] or integer programing [5] to schedule new requests do not take the dynamic nature of the problem into consideration. Testing if a new request can be accommodated essentially involves a rescheduling of the unfinished trips and the new request without reusing the computations in the previous round. Their calculation time was measured in minutes or hours while we require millisecond response time.

1.3 Contributions

To deal with the challenges, our idea is based on a simple observation. For a new valid schedule accommodating the new request tr_i, if we simply drop the three points r_i, s_i, e_i from the trip schedule, then the resulting trip schedule is a valid trip schedule. In other words, only a valid trip schedule can be extended to accommodate a new request. Given this, a potential approach for the ridesharing problem is to simply materialize every valid trip schedule; then, when a new request arrives, we can check if any valid trip schedules can be extended to handle the new request. This approach is promising because its incremental nature saves many redundant computations: We do not need to recompute the valid trip schedule completely from scratch on each new request. However, in order to implement such a strategy, we have to deal with the following challenges: 1) Would the materialization incur too much memory cost? In other words, can we store the materialized schedules compactly? 2) How can we efficiently maintain the materialization? Note that when the server moves, the materialization needs to be updated. 3) How can the materialization help to test quickly whether a new request can be handled? 4) How can the materialization be updated when a new request is accepted?

This paper makes the following contributions:

- We formulate the ridesharing problem in a way that resembles the scenario enabled by current locating and communication technologies; We first propose branch-and-bound and mixed-integer programing algorithms for the problem. We then propose a kinetic tree approach for the problem. The tree structure lends itself naturally to the dynamic nature of the problem;
- When the pickup or dropoff locations are close to each other, any permutation of the locations can be valid, rendering the constraints ineffective and resulting in a large number of valid schedules. We propose a hotspot-based algorithm that ignores schedules that are almost duplicates to effectively reduce the number of valid schedules while providing a bound on the error for the solution under certain conditions;
- We compare our approach to the branch-and-bound and mixed integer programing approaches that are traditionally used, along with the brute-force algorithm. Experiments on a large taxi dataset show that the tree approach is several times to a magnitude faster in response time. We further test tree algorithms on various larger problems to show the performance and effectiveness of the optimizations proposed.

1.4 Outline

We describe the overall ridesharing framework in Section 2. We present a branch-and-bound algorithm and a mixed-integer programming algorithm for scheduling a request in Section 3. We then propose the kinetic tree approach in Section 4. In section 5, we deal with the issue of large trees using a hotspot-based algorithm. Experiment results are presented in Section 6 followed by related work in Section 7 and conclusions in Section 8.

2. FRAMEWORK

When a request is submitted to the system, the request is matched with candidate servers. Because most vehicles will be outside the waiting time constraint w of a trip request at the time that the request is received, we will need a low maintenance cost indexing method to filter out servers outside the waiting time constraint. So, we use a simple grid-based indexing. A grid of uniform cell size $gr.l$ is superimposed on the region, and servers are mapped into the cell corresponding to their current location (the mapping is updated when a server passes between cells).

When a new trip request tr is received in grid cell g, we first calculate the cost of assigning the request to each server within ceil$(\frac{w \cdot D}{gr.l})$ grid cells from g along both axes where D is the speed of server. We then assign tr to the server with the minimum unfinished schedule cost, or permanently reject the request if no server can feasibly handle it. Because the user gets instant feedback, the user may decide to resubmit the request, possibly with relaxed service constraints. We can also allow progressively relaxing the constraints until the request is satisfied.

To calculate the minimum unfinished schedule cost, an instance of the scheduling algorithm is associated with each server. An algorithm instance maintains the corresponding data structure as the server moves along its route in the road network through shortest paths between consecutive points. The shortest paths are calculated and recorded for each server.

Computing shortest path on road networks has been widely studied (see [6] for an extensive review). Recently, Abraham *et al.* [2] discovered that several of the fastest distance computation algorithms need the underlying graphs to have small *highway dimension*. Furthermore, they demonstrate the method with the best time bounds is actually a labeling algorithm [2]. We choose to use the state-of-art hublabeling algorithm - a fast and practical algorithm to heuristically construct the distance labeling on large road networks. After the labeling process, each vertex records a set of intermediate vertices (and distances to them) for the shortest path computation [1]; and they are packed and stored in an array. To answer the distance query from one vertex to another, we perform a linear scan of the labeling arrays of both vertices, and select the minimal distance using all the intermediate vertices existing in both arrays.

The most challenging problem now is: for a given request and a candidate server, find the augmented valid trip schedules in order to find the server with minimum unfinished schedule cost. The algorithms for this problem will be described in the next three section.

3. BRANCH-AND-BOUND AND MIXED INTEGER PROGRAMMING ALGORITHMS

The brute-force algorithm to find the augmented valid trip schedules is straightforward. We enumerate all of the permutations and then check the constraints. However, this can be expensive. Two approaches that are often used in solving the related dial-a-ride problem [9] can increase execution speed: branch-and-bound algorithm [4] and integer programming approach [5]. We first propose a modified branch-and-bound algorithm for our problem, and then formulate the problem as a mixed-integer programming problem.

3.1 Branch and Bound Algorithm

The branch-and-bound algorithm systematically enumerates all candidate schedules and organizes the candidates into a schedule tree. It estimates and maintains a lower bound of each partially constructed schedule and stops building candidate schedules that have lower bounds greater than the best solution found so far. The algorithm first expands the partial candidate with the lowest lower bound (best-first search).

Assume at time t, there are m active trips for the given server. Let the current valid trip schedule be $(x_1, x_2, \cdots, x_{3m})$, where t is between x_c and x_{c+1}. For a new trip request tr_{m+1}, we need to re-schedule the pickup and dropoff points $N = \{x_{c+1}, x_{c+2}, \cdots, x_{3m}, r_{m+1}, s_{m+1}, e_{m+1}\}$. Points in N form a complete graph with edge weights being the shortest path distances between nodes. We attempt to find the schedule through the graph that passes through each node once but, unlike a tour, does not return to the first node. The schedule also has to begin at the location of the server when request r_{m+1} is submitted. In Figure 2 (a), when request r_2 comes in, s_1 is already picked up. So, only $N = \{e_1, s_2, e_2\}$ needs to be scheduled and the schedule must start from r_2.

We start with the initial schedule tree $ST =< r_{m+1} >$, and initialize the cost of the optimal schedule to ∞. We then iteratively

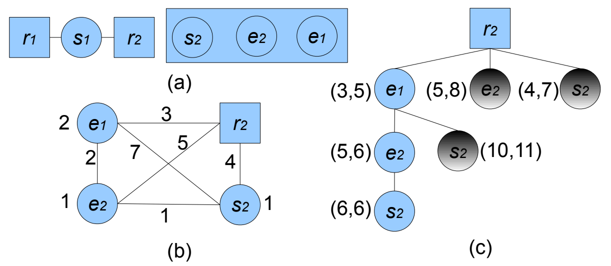

Figure 2: Illustration of Branch-and-Bound Algorithm. (a) When request r_2 comes, only $\{e_1, s_2, e_2\}$ need to be scheduled; (b) Road network distance and minimal incident edge cost; (c) When (r_2, e_1, e_2, s_2) with cost 6 is found, partial schedules with estimated costs above 6 are terminated.

perform a best-first-search to expand the partial schedule $S =< r_{m+1}, x'_{c+1}, x'_{c+2}, \cdots, x'_k >$ with the minimum lower bound. The lower bound we use is $d_T(r_{m+1}, x'_k)$ plus the sum of the costs of the minimum-cost-edges incident to each of the nodes that are not yet in the partial schedule S.

Figure 2 (b) shows road network costs between two nodes. The minimal incident edge cost is labeled beside each node. In Figure 2 (c), for each node x, the two numbers in parentheses indicate the cost $d_T(r_2, x)$ of the partial schedule and the lower bound of the schedule containing the partial schedule as prefix. For (r_2, e_1), $d_T(r_2, e_1) = 3$. Only e_2 and s_2, both with minimal incident edge cost of 1, need to be added to the schedule. The minimal incident edge cost of e_1 is $min(2, 7, 3) = 2$, so the lower bound of a schedule containing (r_2, e_1) is $d_T(r_2, e_1) + 2 = 5$.

We attempt to expand the partial schedule S with minimum lower bound by another new node to construct S'. If S' is not valid or results in a bound greater than the current minimum schedule cost, we terminate S'. If S' is a complete schedule, we compare its cost to that of the best schedule and update if necessary. Once the schedule of cost 6 is found, schedules with lower bounds above 6 can be pruned (labeled by a gray circle). Note that in the figure we do not illustrate validity constraints. The complexity of the branch-and-bound algorithm in the worst case is still exponential.

3.2 Mixed-integer Programming Approach

Mixed integer programing is a popular alternative. In this section, we formulate our augmented valid trip schedule problem into a mixed integer programming problem. Then, we apply traditional solvers to find the solution.

As in the branch-and-bound algorithm, we are rescheduling $N = \{x_{c+1}, x_{c+2}, \cdots, x_{3m}, r_{m+1}, s_{m+1}, e_{m+1}\}$. The schedule must start from r_{m+1}. We divide N into subsets: (1) dropoff locations of those already picked up but not dropped off; let the size of this set be k; (2) pickup locations of trips not started yet; let the size of this set be n; and (3) dropoff locations of trips not started yet; the size of this set is also n. The problem can be defined on a complete directed graph $G' = (N, A)$ where $N = D' \cup P \cup D \cup \{0\}$, $D' = \{1, 2, \ldots, k\}$, $P = \{k+1, k+2, \ldots, k+n\}$, $D = \{k+n+1, k+n+2, \ldots, k+2n\}$. Because of the nature of the problem formulation of integer programing, we abuse the notation of N here: We reshuffle the points in N and assign an integer to each point in N while node 0 represents the current position r_{i+1} of the server. For a pickup i in P, its matching dropoff in D is $i+n$. Each arc $(i, j) \in A$ is associated with a shortest path routing cost d_{ij}. For each arc (i, j), let $y_{ij} = 1$ if the server travels from node i to node j. For each drop point $i \in D' \cup D$, let L_i be the ride time of the request in this partial route. Then, the problem is,

$$Min \sum_{i \in N} \sum_{j \in N} d_{ij} y_{ij}$$

subject to:

$$\begin{array}{lll}
y_{ij} \in \{0,1\}, & \forall i \in N, j \in N & (1) \\
\sum_{j \in N} y_{ji} = 1, & \forall i \in N - \{0\} & (2) \\
\sum_{j \in N} y_{0j} = 1 & & (3) \\
B_0 = 0 & & (4) \\
B_j \geq (B_i + d_{ij}) y_{ij}, & \forall i \in N, j \in N & (5) \\
L_i = B_i - B_{i-n}, & \forall i \in D & (6) \\
B_i \leq w_i, & \forall i \in P & (7) \\
B_i \leq o_i, & \forall i \in D' & (8) \\
d_{i-n,i} \leq L_i \leq \epsilon_i, & \forall i \in D & (9)
\end{array}$$

where w_i is the waiting time left for $i \in P$ and o_i is the maximal riding time left for $i \in D'$. Here d_{ii} is set to a positive number to make sure $y_{ii} = 0$.

The objective is to find the schedule that minimizes the total cost while satisfying the constraints. Constraint (1) simply enforces the binary nature of y_{ij}. Constraint (2) allows exactly one node preceding another for all nodes but 0. Constraint (3) allows exact one node following node 0. These two effectively enforce the schedule structure so that each node is visited exactly once and the schedule starts from node 0.

Constraints (4) and (5) set the earliest time at which a node can be reached. Constraints (6) define L_i for dropoff nodes, the service distance. Constraints (7) and (8) enforce the waiting time and service constraints for pickup and dropoff nodes where the passenger has already been picked up. These are grouped together because both w_i and o_i are measured from the root node. Constraint (9) enforces the service constraint for dropoff nodes where the passenger has not yet been picked up, so that the service time does not exceed ϵ_i. The constraint (5) is not linear. It can be linearized by introducing constants M_{ij}, an approach similar to that in [7].

$$B_j \geq B_i + d_{ij} - M_{ij}(1 - y_{ij}), \forall i \in N, j \in N \qquad (1)$$

The validity of these constraints are ensured by setting $M_{ij} \geq \max\{0, l_i + d_{ij} - e_j\}$ where l_i is the latest time that i needs to be served and e_j is the earliest time that j needs to be served. For $i \in P$, $[e_i, l_i] = [d_{0i}, w_i]$. For $i \in D$, $[e_i, l_i] = [d_{0,i-n} + d_{i_n,i}, w_i + d_{i-n,i}(1+\epsilon)]$. For $i \in D'$, $[e_i, l_i] = [d_{0i}, o_i]$.

Let v be the number of variables in the mixed-integer programming problem, and c be the number of constraints. Then, $v = O(m^2)$ and $c = O(m)$, where m is the total number of requests that we are optimizing.

4. KINETIC TREE APPROACH

The two approaches above both suffer from one fundamental problem: they reschedule unfinished pickups and dropoffs with the new request from scratch, ignoring previous computations. The structures of the two algorithms make it difficult to adapt them to the dynamic nature of the problem. In this section, we introduce a kinetic tree structure that maintains the calculations performed up-to-now and uses them effectively for new requests. However, when there are several pickup or dropoff locations close to each other, the number of feasible schedules increases exponentially. Thus, we propose a hotspot-based approach in Section 5 that reduces the search space and approximates the solution with bounds.

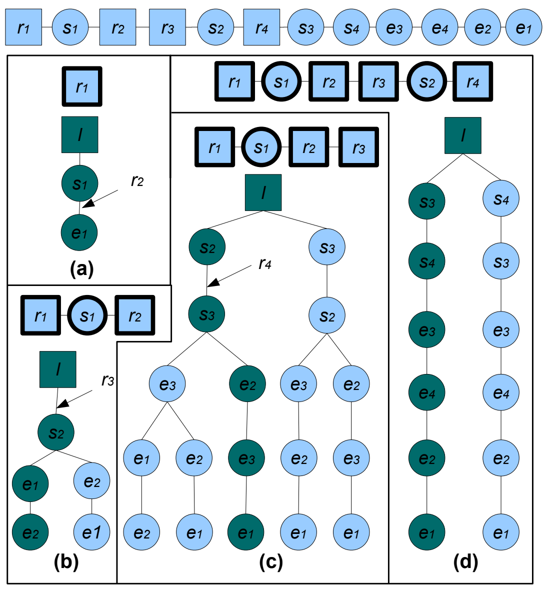

Figure 3: Kinetic Tree for Trip Schedules. Darkened path: selected schedule to be executed; Dark circled/squared nodes: finished nodes.

4.1 Basic Tree Structure

We introduce a kinetic tree structure to maintain all the valid trip schedules with respect to the server's current location. When the server moves, a portion of the schedule becomes obsolete. The root of the tree tracks the current location l of the server. The rest of the tree records portions of all valid schedules (from the current location onwards).

For a given w and ϵ, Figure 3 illustrates the kinetic tree structure corresponding to the complete trip schedule in Figure 1. The darkened path represents the selected (optimal) schedule to be executed by the server. Initially, for the first trip request, there is only one valid trip schedule (Figure 3 (a)). When the second request arrives, the first customer has already been picked up by the server. Now assuming the option which drops the first passenger before picking up the second one is invalid, there are only two valid options for the server to accept the new request: it needs to first pick up the second customer, but it can be flexible in dropping off either of the two passengers. Let us assume it decides to choose the shorter one which is (l, s_2, e_1, e_2), to drop off the first customer first. However, on its way to pick up the second customer, the third request arrives. The server now has the option to pick up either the second customer or the third one. Suppose, based on w and ϵ, that there are five possible valid trip schedules for the server to handle the three trip requests (the first trip is already in progress, shown in Figure 1(c)). Assuming the server decides to move along the shortest route $(l, s_2, s_3, e_2, e_3, e_1)$ for now and picks up the second customer first, then when the fourth request arrives after the pickup of the second customer, the entire right sub-tree of r_3 in Figure 1(c) becomes inactive. Let us now assume there are only two possible schedules to accommodate the remaining trips of all four customers as shown in Figure 1(d).

Why is such a kinetic tree useful in maintaining the valid trip schedules? Its advantage is based on the the following key observation:

LEMMA 1 (VALID SCHEDULES UNDER MOVEMENT). *When* *a server reaches a new pickup or dropoff location in the trip schedule, only those valid schedules which contain unfinished trips and share the same prefix so far (from the first pickup point of all the unfinished schedules to the current location in the trip schedule) need to be materialized. All other schedules become inactive and can be pruned from the tree.*

In Figure 3(c), once the server picks up the second customer, only the schedules in the left sub-tree rooted at s_2 remain active.

4.2 Handling a New Request

Now, we consider how to handle a new request $tr_k = (r_k, s_k, e_k)$. We assume that we already have a materialized prefix tree of all valid and active schedules of unfinished trips. We need to extend all valid and active schedules in the prefix tree to a new valid schedule to include tr_k if possible. We do this by generating a new prefix tree based on the existing one. To deal with the new request, we will first deal with the pickup location s_k and then the dropoff location e_k. Essentially, we need to scan the tree to determine where s_k can be inserted, i.e., which edges of the tree can accommodate the insertion of a new pickup node. All schedules that share the prefix from the root of the tree to the inserted edge will be inserted into the new tree. Then we insert e_k after s_k in the new tree. Furthermore, if s_k or e_k can be inserted at a given location (an edge in the tree), then we have to find out which trip schedules containing that edge with an additional node will become invalid (due to constraint violation) and thus should be pruned. The problem is how to determine 1) at which edge s_k or e_k can be inserted, and 2) how to quickly prune the invalid trip schedules following that insertion.

Inserting Pickup Location: Here, we focus on whether s_k can be inserted first and e_k can be inserted in a similar way later. In order to insert s_k in a tree edge, say (x_i, x_{i+1}), we need to deal with the following situations: (a) only when the distance from the current location (recorded in the root node l) to the pickup location s_k satisfies $d_T(l, s_k) = d(l, x_1) + d(x_1, x_2) + \cdots + d(x_i, s_k) \leq w$, then s_k may be inserted; (b) the additional travel distance (time) introduced by the detour to s_k may invalidate some existing trip schedule in the sub-tree containing this tree edge (x_i, x_{i+1}), i.e., $d(x_i, s_k) + d(s_k, x_{i+1}) - d(x_i, x_{i+1})$ should not be too large. These schedules should be pruned from the sub-tree. Note that condition (a) is easy to be tested in the existing tree structure.

LEMMA 2. *($d_T(l, s_k) \leq w$) The shortest distance from the current location to the requested pickup location s_k is no greater than w. Furthermore, given a prefix (partial) trip schedule from the root node l to a node x_j, i.e., $(l, x_1, x_2, \cdots, x_j)$, if $d_T(l, s_k) = d(l, x_1) + d(x_1, x_2) + \cdots + d(x_j, s_k) > w$, then, any edge incident to any descendant of x_j in the tree cannot accommodate s_k, i.e., the customer cannot wait for the server to finish x_j before being picked up at s_k.*

This lemma suggests that we can perform either a depth first search (DFS) or breadth first search (BFS) starting from the root node of the tree to generate all the candidate edge (x_i, x_{i+1}) to insert s_k. Specifically, during the traversal the visiting will return once certain depth is reached, i.e., a node has the property that $d_T(l, x_j) > w$, in which case we either will not expand that nodes (in BFS) or trace back (in DFS).

Now the key problem is how to handle condition (b). The straightforward way to perform pruning is to explicitly maintain and check constraints for each trip request in the subtree of x_i. Specifically, for a trip tr_j in the sub-tree rooted at x_i, there are two criteria: waiting constraint $[r_j, s_j, w]$ $(d_T(r_j, s_j) \leq w)$ and trip tolerance constraint $[s_j, e_j, \epsilon]$ $(d_T(s_j, e_j) \leq (1+\epsilon)d(s_j, e_j))$. At any given time point t, clearly if we need to test whether the detour meets the criteria of trip tr_j, then the request is already issued and responded, and the entire trip is not yet completed. Furthermore, only one of the criteria needs to be tested: if the server has not picked up the customer, then, we need to test the pickup waiting constraint $[r_j, s_j, w]$; once the customer is picked up, we need to test the trip tolerance constraint $[s_j, e_j, \epsilon]$. Thus, at any given point, the "active" customers can be partitioned into two sets: S_1 records those customers who need to be picked up and S_2 records the on-board customers who need to be dropped off. When a new location is reached, we may move customers from S_1 to S_2 and/or remove customers from S_2. For trip j in S_1, we test the first criterion $[r_j, s_j, w]$ and in S_2, we test the second one: $[s_j, e_j, \epsilon]$. Given this, for the sub-tree rooted at x_i, the simple way is to first generate these two sets S_1 and S_2. Then, when we insert s_k, we need to ensure each condition associated with S_1 and S_2 are also satisfied.

Algorithm 1 insertNodes algorithm.

```
Parameter: root node $l$, request points $P = (x_1, x_2, ...)$, current
    depth $depth$
  if $feasible(l, x_1, depth + d(l, x_1))$ then
    Initialize $fail = 0$
    $n = create(l, x_1)$ {Copy feasible child branches under $n$}
    for each $c$ such that edge $(l, c)$ exists do
      $copyNodes(n, \{c\}, d(l, n) + d(n, c) - d(l, c))$
      If copy failed, set $fail = 1$
    end for{Insert remaining request points to $n$}
    if $fail = 0$ and $|P| > 1$ then {Detour now begins negative
    because we haven't inserted $x_2$ yet}
      $insertNodes(n, \{x_2, ...\}, -d(x_1, x_2))$
      If insert failed, set $fail = 1$
    end if{Now insert request points into children}
    for each $c$ such that edge $(l, c)$ exists do
      $insertNodes(c, P, detour + d(l, c))$
      If insert failed, delete $(l, c)$
    end for
    if $fail = 0$ then
      Add edge $(l, n)$
    else if No nodes $c$ with edge $(l, c)$ exist then
      Insert failed, notify caller that this sub-tree is infeasible
    else
      Insert succeeded
    end if
  else
    Insert failed, notify caller that this sub-tree is infeasible
  end if
```

Algorithm 1 implements the insertion of a new request $tr_k = (s_k, e_k)$ into the tree recursively. The insertion is completed by a call, $insertNodes(root, \{s_k, e_k\}, 0)$. The call to $feasible(parent, node, detour)$ returns whether or not it is feasible to insert $node$ as a child under $parent$ in the tree. First, this ensures that the pickup or service constraint of $node$ is not violated. If min-max filtering is in place (will be discussed), this will confirm that the detour (third argument) is tolerable for $node$.

The $copyNodes(node, source, detour)$ function recursively copies nodes from a set of nodes, $source$, to the target node, $node$. Here, tolerance of the root's children in $insertNodes$ is implemented through calls to $feasible$ with detour of $detour$. $copyNodes$ will fail if all of the children of $node$ are along infeasible paths. In this case, these branches and $node$ will be deleted.

In Figure 4 (a), we use the insertion algorithm to insert the pickup location s_3 into an existing tree, thereby generating a new tree. s_3 will first be inserted directly below l. Then, the branch with root at s_2 will be copied underneath this new s_3 node, forming

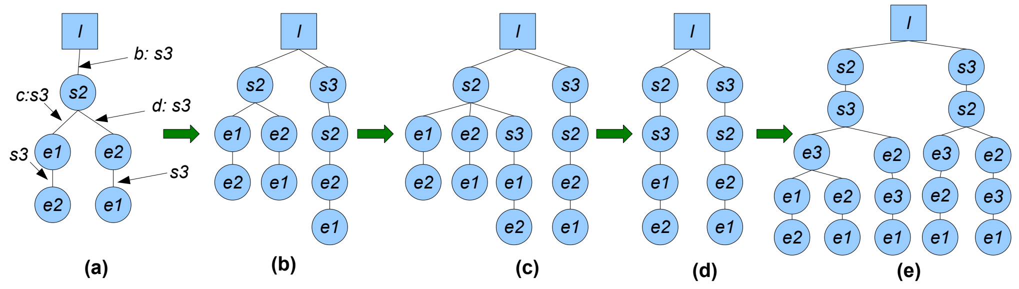

Figure 4: Tree Insertion. The insertions of s_3 into each edge in tree of (a) result in (b), (c), (d), and (e), assuming the last two insertions were infeasible.

a new tree of $(l, s_3, s_2, ((e_1, e_2), (e_2, e_1)))$. Let us assume route (l, s_3, s_2, e_1, e_2) is not feasible; then, the branch is pruned from the tree starting at the leaf node until we reach s_2, where we have an alternate path (l, s_3, s_2, e_2, e_1) that is feasible. This pruning occurs in the $copyNodes$ algorithm, which will succeed because s_3 falls along at least one (in this case, exactly one) feasible path. The resulting tree is shown in Figure 4 (b).

Then, the insertion algorithm moves down to s_2 and attempts to insert the pickup location after it. Two paths are formed: (l, s_2, s_3, e_1, e_2) and (l, s_2, s_3, e_2, e_1), as a result of the insertion between s_2 and e_2 and between s_2 and e_1. Suppose (l, s_2, s_3, e_2, e_1) is infeasible. The resulting tree is shown in Figure 4 (c). Suppose inserting s_3 between e_1 and e_2 or between e_2 and e_1 is infeasible. Then, we have the tree in Figure 4 (d). To complete the insertion of the (s_3, e_3), we now try to insert e_3 in the sub-trees that root at a s_3 following the same insertion algorithm. Once this completes, we arrive at the tree shown in Figure 4 (e).

Min-max Filtering using Slack time: Though the above test for condition (b) is conceptually simple, it is rather computationally expensive. Now, we introduce a fast approach to simplify and speedup such test. For any node j, if j is in S_1, let $\delta_j = w - d_T(r_j, s_j)$; otherwise ($j$ is in S_2), let $\delta_j = (1 + \epsilon)d(s_j, e_j) - d_T(s_j, e_j)$. Then, for the node x_j, we associate **slack time** $\Delta_{x_j} = \min(\delta_j, \max_{i \in x_j.children} \Delta_i)$.

Note that Δ_{x_j} essentially represents the minimal allowed detour on the most "lenient" route of the subtree routed at x_j. Here "lenient" means the route can tolerate the most detour compared to other routes. Given this, we introduce the following Theorem to describe the simple condition to determine whether s_k can be inserted at a given edge.

THEOREM 1. *For a trip request tr_k, if edge (x_i, x_{i+1}) does not satisfy either of the following condition: (a) $d_T(l, s_k) = d(l, x_1) + d(x_1, x_2) + \cdots + d(x_i, s_k) \leq w$; or (b) $d(x_i, s_k) + d(s_k, x_{i+1}) - d(x_i, x_{i+1}) \leq \Delta_{(x_i, x_{i+1})}$, then, we can not add the pickup s_k between location x_i and x_{i+1}.*

After insertion in (x_i, x_{i+1}), all nodes under x_i of the new tree will be tested for the constraint $\delta_i \geq d(x_i, s_k) + d(s_k, x_{i+1}) - d(x_i, x_{i+1})$. A branch is pruned from the sub-tree if the constraint is not satisfied.

Updating Δ and Tree: After we try to insert a request to all possible servers, we get a set of new trees. For each tree, we can find the shortest route and choose the tree that provides the shortest route among all trees. Only the chosen tree needs to have its Δ updated. This can be done through one tree traversal. When a server is moving, the tree needs to be updated as well. However, the Δ values are quiescent to server movement and do not need to be updated. The tree is updated when a vehicle reaches a new pickup or dropoff location; the server drops the inactive portion of the tree accordingly.

Practical concerns: In practice, customers may occasionally wish to cancel a trip after it has been scheduled. To deal with this, we can reconstruct the tree from scratch without the canceled trip request to derive an optimal solution. Alternatively, we can derive an approximate solution by simply deleting all instances of the canceled pickup and dropoff location from the current tree, and selecting the shortest resulting route; while this ignores branches that become feasible following the deletion, it still produces better solutions than simply deleting the customer from only the current route.

Another practical issue is unexpected traffic conditions. If a vehicle runs into unexpected delay, then although it does still have to drop off on-board passengers regardless of service constraints, we can cancel and possibly re-assign the trips that have not started yet to another vehicle. To determine if constraints are violated, we simply compare the additional delay encountered to the slack time along the optimal branch. The trips with waiting time constraint being violated will be iteratively canceled and re-assigned which may also lead to better service for on-board passengers.

5. HOTSPOT BASED OPTIMIZATION

The main problem with the basic tree algorithm is the exponential explosion of the size of the tree when there are multiple pickup or dropoff locations close to each other. For example, if 8 pickups occur in spatial-temporal proximity, e.g an airport after a flight lands, any permutation of the pickups may result in a valid schedule. This yields $8! = 40,320$ possibilities, even before considering dropoff points. We propose an approximation approach with bound to reduce the search space. The idea is that when the time and space requirements of computing the best schedule are too high, a server may decide to reduce the load by only maintaining a representative subset of the schedules. Since the number of leaves of the kinetic tree is determined by the number of possible routes, the tree size is effectively controlled by the approximation and the service constraints.

We propose the following **hotspot clustering algorithm** to deal with this situation. When we insert a pickup point s_k to an edge (x_i, x_{i+1}), we check if $d(x_{i+1}, s_k) \leq \theta$ for small θ. If so, s_k is inserted into the node of x_{i+1}. s_k and x_{i+1} are treated as one point called a *hotspot* in the tree, and an arbitrary schedule is chosen between the points in a hotspot. When a hotspot contains more than one point, a newly inserted point must be within θ to all points of

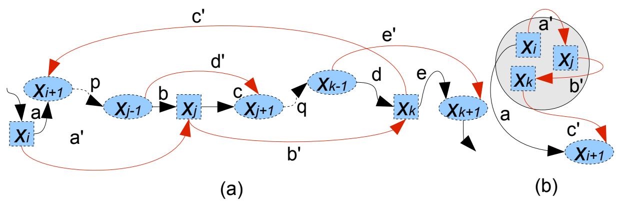

Figure 5: Bound for Hotspot. x_i, x_j, and x_k are in one hotspot. Black lines: optimal schedule S_{best}. We can convert S_{best} by connecting x_i, x_j, x_k consecutively first and then thread the other locations (represented by ovals). The new schedule has a bounded cost.

the hotspot. A similar procedure can be followed for dropoff points and mixtures of pickup and dropoffs. Once a point is combined with any hotspot, we stop trying to insert it to any other edges.

Let us first assume the service constraints are sufficiently large that all schedules are possible. For a trip set TR, let S_{best} be the optimal schedule. Suppose there is a hotspot hp among the pickup and dropoff locations of TR. Our hotspot-based method chooses an arbitrary schedule T_{hs} that goes through the points of the hotspot in a consecutive manner. We want to prove that the cost of T_{hs} is bounded.

THEOREM 2. *$cost(T_{hs}) \leq cost(S_{best}) + (2m+1) \times \theta$ where m is the number of points in the hotspot without considering constraints.*

Proof Sketch: We prove when $m = 3$ by illustration. For general m, the proof is mainly the same. In Figure 5 (a), assume $\{x_i, x_j, x_k\}$ has pairwise distance of no greater than θ. The optimal schedule S_{best} is labeled by black solid and dashed lines. We can convert S_{best} into T_{hs} by connecting x_i, x_j, x_k consecutively first and then thread the other locations (represented by ovals) in S_{best} as shown by the red lines and black dashed lines. We prove that (1) $cost(T_{hs}) \leq cost(S_{best}) + 3\theta$ which is equivalent to prove $a' + b' + c' + d' + e' \leq a + b + c + d + e + 3 \times \theta$ since the dashed lines are common in both schedules.

We know $d' \leq b + c$, $e' < d + e$. Now we only need to show $a' + b' + c' \leq a + 3 \times \theta$. As shown in 5 (b), we can easily prove that $c' \leq a + \theta$ because the shortest path between x_k and x_{i+1} is no longer than than the schedule x_k, x_i, x_{i+1}. Because $a' \leq \theta$ and $b' \leq \theta$, we know $a' + b' + c' \leq a + 3 \times \theta$.

However, the hotspot algorithm may not use the same order of x_i, x_j, x_k as in the optimal solution as it is an arbitrary order, we now prove that (2) for any two hotspot-based schedule S_{hs} and $S_{hs'}$, $cost(S_{hs}) \leq cost(S_{hs'}) + (m+1)\theta$ where $m = 3$. Without loss of generality, let $S_{hs} = \ldots, x_{i-1}, x_i, x_j, x_k, x_{k+1} \ldots$ and $S_{hs'} = \ldots, x_{i-1}, x_j, x_k, x_i, x_{k+1} \ldots$. It is obvious that $d(x_{i-1}, x_i) \leq d(x_{i-1}, x_j) + \theta$ and $d(x_k, x_{k+1}) \leq d(x_i, x_{k+1}) + \theta$. Also $d(x_i, x_j) \leq d(x_j, x_k) + \theta$ and $d(x_j, x_k) \leq d(x_k, x_i) + \theta$. Adding the inequalities together, we have $cost(S_{hs}) \leq cost(S_{hs'}) + 4\theta$

Putting (1) and (2) together, we have $cost(S_{hs}) \leq cost(S_{best}) + (2m+1) \times \theta$ where $m = 3$. □

Because after we build the whole tree, we select the shortest schedule with hotspot $cost(S_{hsBest})$ and it is obvious that $cost(S_{hsBest}) \leq cost(S_{best}) + (2m+1) \times \theta$.

When we consider the constraints, for S_{best} the corresponding hotspot-based schedule with constraint may violate some constraints and thus does not exist. However, when the constraints of points of the best schedule are relaxed, the corresponding hotspot-based schedule will be found. We have the following theorem.

THEOREM 3. *$cost(S_{hs}) \leq cost(S_{best}) + (2m+1) \times \theta$ where m is the number of points in the hotspot when constraints of all points in S_{best} is larger than $m\theta$.*

Proof Sketch: Again we prove for $m = 3$ because of the ease of illustration. In Figure 5 (a), if (a, p, b, c, q, d, e) is a valid partial schedule with each node having at least 3θ slack time, then $(a', b', c', p, d', q, e')$ is a valid partial schedule.

For any point on p, the extra delay is $a' + b' + c' - a \leq 3 \times \theta$. For any point on q, the extra delay $a' + b' + c' - a + d' - (b + c) \leq a' + b' + c' \leq 3 \times \theta$. For x_{i+1}, the extra delay is $a' + b' + c' + d' + p + q - (a + b + c + d + p + q)$ which is proven in Theorem 2 as no larger than $3 \times \theta$. □

When θ is sufficiently small, we will likely to find a schedule that is upper bounded by the best schedule with a small additional time.

6. EXPERIMENTAL DESIGN

We evaluate the algorithms using a large scale taxi dataset containing **432,327 trips** made by **17,000 Shanghai taxis** over one day (May 29, 2009). A trip tp in the dataset has starting time $tp.st$, starting location $tp.sl$, ending time $tp.et$, and ending location $tp.el$. A simulator generates trip requests from the actual 432,327 trips and submits them to the scheduling system in real-time. Specifically, for each trip tp, a trip request tr is initialized as $tr = \langle tp.sl, tp.el, w, \epsilon \rangle$, and tr is submitted at time $tp.st$.

The Shanghai road network is represented by an undirected and weighted graph containing 122,319 vertices and 188,426 edges. The starting and ending trip coordinates are pre-mapped to the closest vertex in the graph. The road network is stored in memory in a simple weighted adjacency list structure.

A vehicle is initialized to a random vertex in the road network. Vehicles follow a given route when customers are on board or, otherwise, follow the current road segment, choosing a random segment to follow at intersections. We assume a constant speed D; specifically, based on the data, we set D to 48 km/hr. Time-distance conversion is accomplished by multiplying or dividing by D. In our experiments, the superimposed grid size $gr.l$ is 500 m.

For large scale ridesharing, the shortest path algorithm is called very frequently. We observe the repeated calling from scheduling algorithms follows a pattern that preserves locality. We implement two **LRU caches**: one storing up to ten million shortest distances and the other storing up to ten thousand shortest paths. A new shortest distance/path is calculated when there is a cache miss and will replace the least recently used one. Both caches are indexed only by the starting and ending points in a distance or path computation call; this is accomplished by defining the index for two vertices s and e as $i = \text{id}(s) \cdot |V| + \text{id}(e)$, where $|V|$ is the total number of vertices and id returns an integer representation for a vertex.

The simulation framework is implemented in C++. We run the experiments on cluster nodes with an Intel Xeon X5550 (2.67GHz) processor. The simulation implementation is single-threaded, and memory usage is limited to three gigabytes.

Parameter	Tested settings
Scheduling Capacity	**4**
Constraints	5 min / 10%; **10 min / 20%**; 15 min / 30%; 20 min / 40%; 25 min / 50%
Number of Servers	1,000; 2,000; 5,000; **10,000**; 20,000

Table 1: Parameters for four-algorithm comparison.

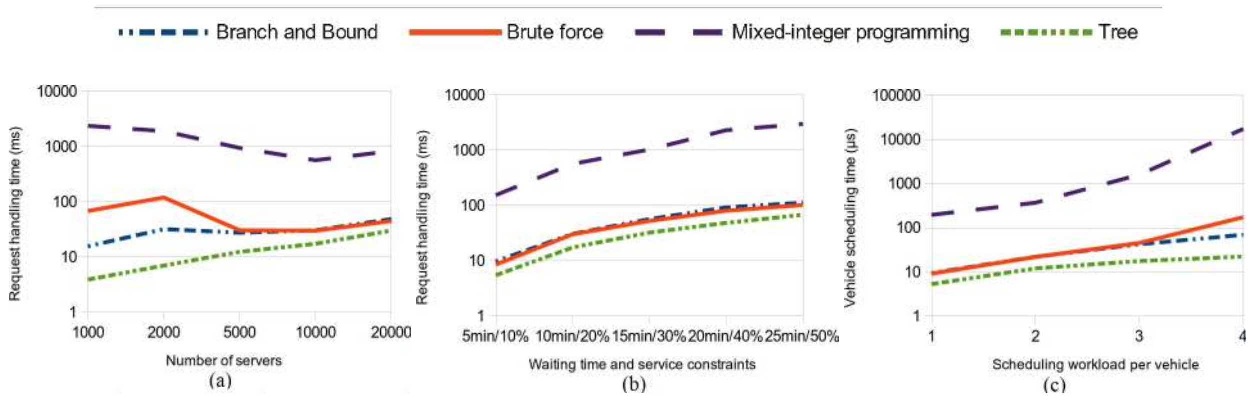

Figure 6: Four algorithm Comparison. (a) Average RHT with respect to number of servers; (b) Average RHT with respect to constraints; (c) Average VST with respect to scheduling workload per vehicle. Default parameters are 10 min / 20% for the constraints, 10,000 servers, and a scheduling capacity of 4.

6.1 Four Algorithm Comparison

We first compare kinetic tree algorithm with the branch-and-bound, brute-force, and mixed-integer programming algorithms. We choose three important parameters: scheduling capacity, waiting time and service constraints, and number of servers. We first establish reasonable defaults for the parameters, and then proceed to modify the parameters one at a time to evaluate their effect. The defaults (bolded) and other tested settings are shown in Table 1. Because only our kinetic tree algorithms respond in real-time for higher scheduling capacities, the default scheduling capacity is set at 4 for experiments (which also mimic real-world taxi system) in this section. We test much larger scheduling capacities for kinetic tree algorithms in section 6.2. Note that a waiting time constraint of 10 minutes corresponds to 8,000 meters.

To validate our settings for the number of servers, we run the simulation first with different numbers of servers and evaluate the drop rate (i.e., the number of trip requests that are rejected by the system as a percentage of total requests) at each setting. These results are shown in Table 2. As can be seen, there is a steep decrease in the drop rate from 2,000 to 5,000 and again from 5,000 to 10,000 servers, making either 5,000 or 10,000 a good choice for this data (assuming we want to satisfy most requests). It is also interesting that drop rates are similar between the two scheduling capacities; this is because most servers are not filled to capacity.

#servers	Scheduling capacity 4	Scheduling capacity 6
500	84.0%	84.2%
1000	72.1%	72.1%
2000	50.4%	51.3%
5000	8.52%	7.02%
10000	$< 0.01\%$	$< 0.01\%$
20000	0%	0%

Table 2: Drop rate evaluation. The percentage of unsatisfied customers (customers whose requests were denied because no server was available) is shown for constraints of 10 minutes and 20%, and variable number of servers and scheduling capacity.

To evaluate performance, we measure the vehicle scheduling time (VST): the time needed to attempt to schedule a trip request to a vehicle given its current state, i.e., to calculate the minimum schedule cost for the active trips and the new request. Depending on the scheduling workload, the VST can change significantly (for example, a taxi with twenty active requests would have forty more points to be scheduled than one with no assigned requests). Thus, we show average VST for different scheduling workload.

Because a request will need to be matched to all vehicles within w to pick the best, we also measure the request handling time (RHT): the time required to search for and assign the new request to the vehicle with minimum cost or to reject it (RHT includes VST for each vehicle searched).

Figure 6 (a) and (b) show the average RHT with varying fleet size and constraints and Figure 6 (c) shows the average VST with respect to different scheduling workload per vehicle. Generally, the brute-force and branch-and-bound algorithms exhibit roughly the same performance. The mixed-integer programming approach takes significantly more time, probably because of significant execution time used to initialize and preprocess each mixed-integer programming problem. The tree algorithm outperforms the other algorithms for all test cases often by orders of magnitude, due to its incremental approach.

For a small number of taxis and a large scheduling workload, branch-and-bound outperforms brute-force. The reason is most likely that the pruning effect of branch-and-bound is more important when scheduling more requests. When the problem size is small, the fast initialization of the brute-force algorithm is preferable.

For the default parameters, the execution time of the branch-and-bound and brute-force algorithms are almost identical, while the mixed-integer programming is approximately 20 times slower. The tree algorithm, on the other hand, is almost two times faster than the branch-and-bound algorithm. Similar magnitude execution time differences are seen for other parameters.When examining the vehicle scheduling time only for computations where the scheduling workload is 4 (at the capacity), the tree algorithm becomes 5 times to several orders of magnitude faster than the other three algorithms.

6.2 Comparing Tree Algorithms

We further evaluate different versions of our tree algorithm on parameters that the other algorithms cannot efficiently handle: basic tree algorithm, the slack time algorithm, and the hotspot clustering algorithm (which also uses slack time).

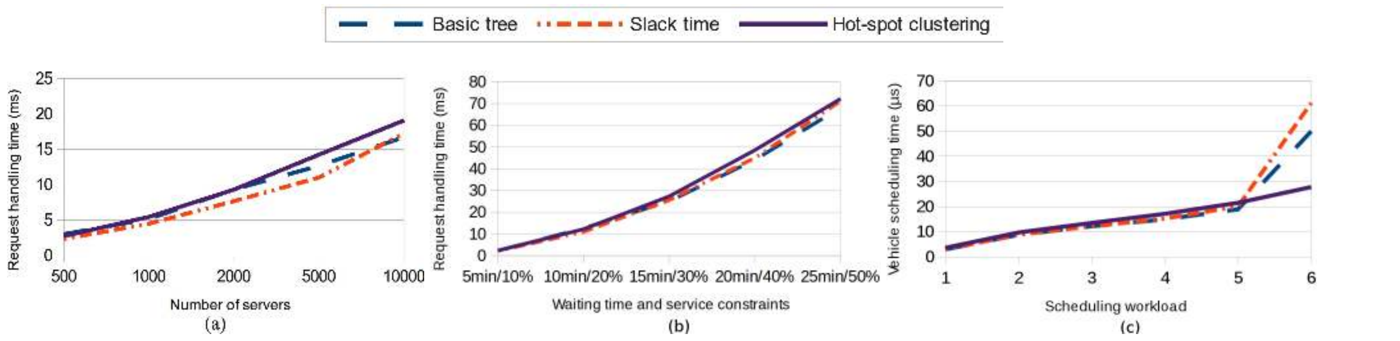

Figure 7: Average performance of tree algorithms. (a) RHT with respect to number of servers; (b) RHT with respect to constraints; (c) VST with respect to scheduling workload per vehicle. Default parameters are 10 min / 20% for the constraints, 5,000 servers, and a scheduling capacity of 6.

Parameter	Tested settings
Scheduling Capacity	3; 4; 5; **6**; 7; 8; 12; 16; unlimited
Number of Servers	500; 1000; 2000; **5,000**; 10,000
Constraints	5 min / 10%; **10 min / 20%**; 15 min / 30%; 20 min / 40%; 25 min / 50%

Table 3: Parameters for Tree algorithm Comparison.

Because only the hotspot clustering algorithm can handle an unlimited scheduling capacity (results shown in Figure 9), we set a default scheduling capacity of 6 for comparison. The parameters we use for evaluating the tree algorithms are shown in Table 3, with the bold values being the default settings. Also, for the hotspot clustering algorithm, we fix the hotspot combination threshold θ to 50 meters.

Figure 7 shows that slack-time algorithm generally has a higher performance in request handling time than the basic tree algorithm for the lower three constraint values tested. This makes sense because the slack-time algorithm is designed to improve the detection of infeasible branches; when constraints are tighter, fast detection is most useful.

Figure 8 presents VST with a scheduling workload of 6. Most prominent in the graphs is the steep increase in VST for tight constraints and large number of servers with the basic and slack-time tree algorithms. In both of these cases, it is relatively rare for a server to have six passengers: typically, there would either be another server with less passengers available to handle the request or the constraints would be too tight to allow so many passengers. So, when the server is able to get six passengers, it is most likely because the pickup/dropff points are very close to each other. In these cases, the short distance between the points creates a large number of feasible combinations. Although these cases would also appear for looser constraints and smaller numbers of servers, the VST is an average, and other six-passenger-cases that do not create a large number of combinations would be much more common. This also explains why the hotspot clustering algorithm is not affected by the trend. In Figure 8 (b), with very high numbers of servers, some of the clustered trips will be split between servers since it is more likely that multiple servers will be nearby; although this trend generally affects non-clustered trips to a greater degree (thus explaining the increase in VST from 2,000 to 5,000 in the first place), there are few enough cases where we get to the scheduling workload of six, that fluctuations in the taxi positions as they are serving passengers or cruising greatly affect the performance. Slack-time approach is designed to handle cases where there are many possibilities that can be pruned; when this is not the case it adds unnecessary overhead.

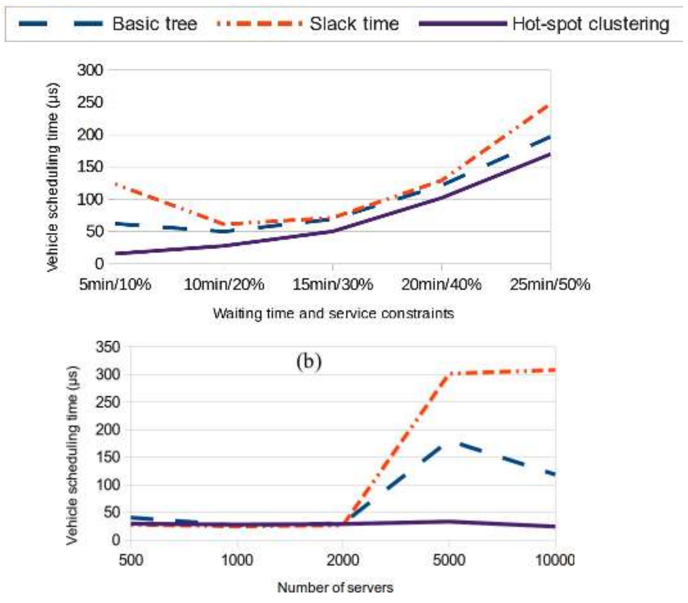

Figure 8: Tree algorithm comparison for scheduling workload of 6 only. (a) Average VST with respect to constraints; (b) Average VST with respect to number of servers. Default parameters are constraints of 10 min / 20% and scheduling capacity of 6 with 5,000 servers.

In these extreme cases, the slack-time tree algorithm is slower than the basic tree algorithm; slack time only reduces execution time when there are many infeasible branches that can be pruned, but here most of the branches remain feasible. Slack time retains its usefulness at smaller scheduling workloads, where it is scalable across other parameters. Since the hotspot clustering algorithm also utilizes a slack-time based approach, it gets advantages at both low and high scheduling workloads.

Figure 9 shows RHT for increasing scheduling capacity. The RHT breaks off for each algorithm when it can no longer finish in a reasonable time or exceeds the 3GB memory limit. The hotspot clustering algorithm is the only one that is able to finish and response in real-time with a capacity greater than 7, and also for unlimited scheduling capacity (marked as unlim in the figure).

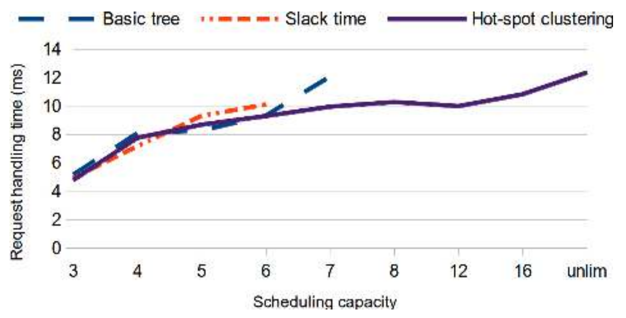

Figure 9: Average RHT for different scheduling capacities. "unlim" indicates unlimited capacity. Only hotspot clustering algorithm can complete for unlimited capacity. Default parameters are 10 min / 20% for the constraints and 5,000 servers.

From this figure, we can see that while the basic and slack-time tree algorithms are unable to continue processing when the problem sizes become too large, the hotspot clustering algorithm is scalable to higher capacities. This also confirms our hypothesis that when a large number of passengers wish to depart from a single point (exponential scheduling possibilities), hotspot clustering combines these points in the tree.

Our experiments above on the Shanghai dataset show that the maximum number of passengers at unlimited capacity in a single taxi is 21, while the average is 0.8 (this is with the default parameters, so with 5,000 servers; at 2,000 servers and unlimited capacity, the average bumps up to 1.7). The average in the top 20% filled taxis is 2.8 (3.9 with 2,000 servers).

To get an idea of the effect of different θ parameters on the quality of solutions produced by the hot-spot clustering algorithm, we test $\theta = 50, 200, 500$ meters. Specifically, we record the percent increase in average distance travelled by a satisfied trip (ADT) from basic tree algorithm to hotspot-clustering algorithm; this shows how much extra distance needs to be covered to satisfy a similar amount of requests due to the clustering. Figure 10 shows that when θ increases, the ADT does not always increase. This is because along with degraded solution quality, higher θ values also may fail to satisfy the service constraints exactly as nodes are merged; so, more customers can actually be satisfied. Still, particularly for higher numbers of servers, the raw trip data shows that the hotspot-cluster algorithm does yield similar solutions as the other algorithms with $\theta = 50$ meters.

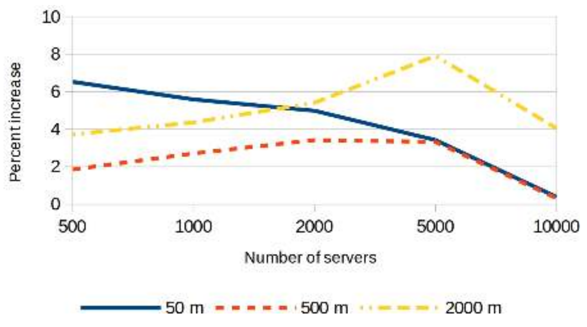

Figure 10: Percent increase in average distance travelled by a satisfied trip from basic tree algorithm to hotspot-clustering algorithm with varying θ and number of servers. Higher numbers indicate degraded solution quality.

We additionally conduct an experiment to explore a case where we never reject customers as shown in table 4. To achieve this, we modify the algorithms to support request-specific pickup and service constraints (this is a simple modification). Then, when a new request is received, we first attempt to schedule it with the tightest configured constraints (5 minutes, 10%), then try the next looser constraint values (factoring the contraints by $1.0, 1.5, 2.0, 2.5, 2.5*1.5$, and $2.5*1.5^2$) until the request is scheduled. Table 4 shows the average constraints across all requests when using this approach. The results show that, similar to drop rate, there is a significant reduction in average constraints from 2,000 servers to 5,000 servers.

Number of servers	Pickup constraint	Service constraint
500	56.7 minutes	113.4%
1000	45 minutes	90%
2000	35.1 minutes	70.2%
5000	10.6 minutes	21.4%
10000	10 minutes	20%

Table 4: Average constraints across customers when all requests are satisfied by increasing the service constraints until a server can handle the request.

7. RELATED WORK

Our work is related to nearest neighbor (NN) search on moving objects over road networks. Early work focuses on data models that are easy to implement and serve as a foundation for NN queries [14]. Later research has focused on continuous monitoring of NNs in highly dynamic scenarios, where the queries and the data objects move frequently on a road network [19]. A recent paper addresses the problem of monitoring the k-NN to a dynamically changing path in road networks. Given a destination where a user is going to, this new query returns the k-NN with respect to the shortest path connecting the destination and the user's current location [3]. Guting *et. al.* proposed algorithms to find the k-NNs to m_q within D for any instant of time within the lifetime of m_q given a set of moving object trajectories D and a query trajectory m_q [12]. NN query in road network is orthogonal to ridesharing scheduling problem that can help to filter the initial set of candidate taxis.

The *trip grouping* algorithm [11] groups "closeby" requests using a set of heuristics. Requests are queued for a waiting time to be scheduled. The heuristics include grouping requests upon expiration, estimation combination saving using pairwise request combination gain, and greedy grouping. The grouping algorithm is then expressed as a continuous stream query and optimized by space partitioning and parallelization. This method is heuristic-based and does not provide waiting and riding time service guarantee as our method does. A recent paper [17] formulated the ridesharing problem similarly (early online version of our paper is available from [13]). However, their work focuses on the effect of indexing and approximate routes on level of ridesharing and satisfaction rate while ours focuses on efficient and effective algorithms for optimal scheduling. Particularly, for any taxi, our solution can guarantee to find the optimal route given the existing requests, while [17] cannot provide this as their solution consider only one greedy route for the requests. We have studied both branch-and-bound algorithms and the novel kinetic tree approach. The matching algorithm in [17] can be considered as a special case of the kinetic tree approach where only one branch is recorded.

In operation research, early research on this problem mostly focuses on a single vehicle and a static scenario where the set of requests are known ahead of time. This is unrealistic for large scale and ad-hoc services such as a taxi service. The problem is, unsurprisingly, NP-hard. Only problems with small sizes can be solved to optimality. Exact dynamic programming algorithms have been developed [20]. However, in our case, since the maximal waiting time and the service level are two separate constraints, each trip request cannot be enforced with a fixed completion deadline. Thus, the dynamic programming approaches can not be applied. We also note that our problem can be considered more general than the fixed deadline problem. Given a fixed deadline t, the maximal waiting time can be defined as $w = t - (1+\epsilon)d(s,e)$. Thus, our algorithm can also be used for the fixed deadline problem.

In a dynamic single vehicle DARP problem, requests come in real time and a server has to make decisions on-line [9]. In the problems without deadline, the objectives are to minimize *makespan* (time to finish the last request) or the *average completion time*. *Competitive ratio* is a standard tool to measure the effectiveness of a dynamic DARP algorithm. An on-line algorithm A is called $c - competitive$ if for any instance δ, the cost of A on δ is at most c times the offline optimum on δ. This is assuming an optimal solution is available which is false for modern large scale scheduling problem we are addressing .

The state-of-the-art Branch-and-cut (*BaC*) algorithm [5] formulates the multiple server version of this problem using mixed-integer programing and a branch-and-cut solution. *BaC* can find exact solutions for small to medium size instances (4 vehicle and 32 requests on a moderate PC for tens to hundreds of minutes). It assumes all vehicles and requests are available ahead of time which is not realistic for a dynamic taxi service of thousands of vehicles serving through out the day. Nevertheless, the solution can be adopted to accommodate the attempts of combining new requests with existing routes of vehicles. We compare our kinetic tree based approach to a branch-and-bound approach and a mixed integer programing approach in this paper.

8. CONCLUSION

In this paper, we formulate and propose a branch-and-bound algorithm, a mixed-integer-programing based algorithm, and a kinetic tree algorithm with optimizations to dynamically match real-time trip requests to servers in a road network to allow ridesharing. The proposed kinetic tree algorithm outperforms commonly used approaches including branch-and-bound and mixed-integer programing. Experiments on large taxi datasets show the advantages of the kinetic tree approach. In the future, we would like to consider uncertainty introduced by traffic and pick-up/drop-off delays in scheduling; such uncertainty is inherent to ridesharing. A model that captures and informs users about the uncertain nature of the scheduling allows users to make appropriate choices based on their individual needs.

9. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the 10th Intl. Conf. on Experimental algorithms*, 2011.

[2] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *SODA*, 2010.

[3] Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring path nearest neighbor in road networks. In *SIGMOD*, pages 591–602, 2009.

[4] A. Colorni and G. Righini. Modeling and optimizing dynamic dial-a-ride problems. *International Transaction in Operation Research*, 8(2):156–166, 2001.

[5] J.-F. Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operation Research*, 54(3):573–586, 2006.

[6] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Algorithmics of large and complex networks. chapter Engineering Route Planning Algorithms. 2009.

[7] M. Desrochers and G. Laporte. Improvements and extensions to the miller-tucker-zemlin subtour elimination constraints. *Operations Research Letters*, 36(10):27–36, 1991.

[8] J. F. Dillenburg, O. Wolfson, and P. C. Nelson. The intelligent travel assistant. In *The IEEE 5th International Conference on Intelligent Transportation Systems*, pages 691–696, 2002.

[9] E. Feuerstein and L. Stougie. On-line single-server dial-a-ride problems. *Theoretical Computer Science*, 268(1):91–105, Oct. 2001.

[10] K. Ghoseiri, A. Haghani, and M. Hamedi. Real-time rideshare matching problem. *Final Report of UMD-2009-05, U.S. Department of Transportation*, 2011.

[11] G. Gidofalvi, T. B. Pedersen, T. Risch, and E. Zeitler. Highly scalable trip grouping for large-scale collective transportation systems. In *International Conference on Extending Database Technology,*, pages 678–689, 2008.

[12] R. H. Güting, T. Behr, and J. Xu. Efficient k-nearest neighbor search on moving object trajectories. *The VLDB Journal*, 19(5):687–714, Oct. 2010.

[13] Y. Huang, R. Jin, F. Bastani, and X. S. Wang. Large scale real-time ridesharing with service guarantee on road networks. *CoRR*, abs/1302.6666, 2013.

[14] C. S. Jensen, J. Kolářvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *ACM GIS*, pages 1–8, 2003.

[15] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *Very Large Databases (VLDB)*, pages 768–779, 2004.

[16] 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*, 22(3):377–386, 1985.

[17] S. Ma, Y. Zheng, and O. Wolfson. T-share: A large-scale dynamic taxi ridesharing service. In *ICDE*, pages 410–421, 2013.

[18] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.

[19] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *Very Large Databases (VLDB)*, pages 43–54, 2006.

[20] H. Psaraftis. An exact algorithm for the single-vehicle many-to-many dial-a-ride problem with time windows. *Transportation Science*, 17(3):351–357, 1983.

[21] Y. Tao, D. Papadias, and J. Sun. The tpr*-tree: an optimized spatio-temporal access method for predictive queries. In *Very Large Databases (VLDB)*, pages 790–801, 2003.

[22] C. Tian, Y. Huang, Z. Liu, F. Bastani, and R. Jin. Noah: a dynamic ridesharing system. In *SIGMOD Conference, Demo*, pages 985–988, 2013.

[23] TICKENGO. Tickengo. http://tickengo.com.