Ridesharing: Simulator, Benchmark, and Evaluation

James J. Pan
Tsinghua University, China
pan-j16@mails.tsinghua.edu.cn

Guoliang Li
Tsinghua University, China
liguoliang@mail.tsinghua.edu.cn

Juntao Hu
Beihang University, China
hujuntao@buaa.edu.cn

ABSTRACT

Ridesharing is becoming a popular mode of transportation with profound effects on the industry. Recent algorithms for vehicle-to-customer matching have been developed; yet cross-study evaluations of their performance and applicability to real-world ridesharing are lacking. Evaluation is complicated by the online and real-time nature of the ridesharing problem. In this paper, we develop a simulator for evaluating ridesharing algorithms, and we provide a set of benchmarks to test a wide range of scenarios encountered in the real world. These scenarios include different road networks, different numbers of vehicles, larger scales of customer requests, and others. We apply the benchmarks to several state-of-the-art search and join based ridesharing algorithms to demonstrate the usefulness of the simulator and the benchmarks. We find quickly-computable heuristics outperforming other more complex methods, primarily due to faster computation speed. Our work points the direction for designing and evaluating future ridesharing algorithms.

PVLDB Reference Format:
James J. Pan, Guoliang Li, Juntao Hu. Ridesharing: Simulator, Benchmark, and Experimental Evaluation. *PVLDB*, 12(10): 1085–1098, 2019.
DOI: https://doi.org/10.14778/3339490.3339493

1. INTRODUCTION

Ridesharing is an emerging mode of transportation currently having a deep impact on the personal transportion industry. The basic concept is that customer passengers needing transport can hail participating vehicles on their smartphones. These vehicles can serve multiple passengers at a time in order to take advantage of similar trips to save travel distance and fuel cost. Figure 1 gives an example.

Commercial services today are already facing enormous customer loads, on the order of millions per day [2]. Societal benefits from ridesharing [40, 10, 24] can be captured by optimizing assignment of customers to vehicles so as to maximize utilization of the vehicles while simultaneously using least possible travel distance. These assignment decisions can be characterized by the speed of the decision-making algorithm and the quality of the assignment toward optimization objectives. Despite studies on assignment approaches from the database [11, 12, 41, 36, 37, 9, 27, 20] and other communities [6, 29, 22, 16, 34, 5, 15], challenges remain.

First, most evaluations do not consider the online and real-time nature of ridesharing, possibly because offline evaluation is much simpler. But these offline evaluations do not reflect the real world. Thus, there is a need for an *online and real-time* evaluation platform.

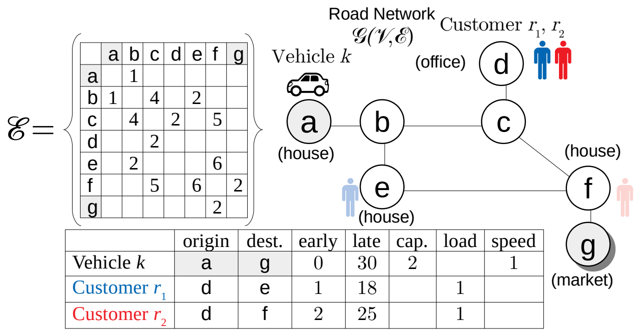
Second, there is no consistent benchmark for ridesharing, needed due to huge variation in problem scenarios. Algorithm performance can be affected by many factors: the road network; volume and velocity of customer and vehicle streams; properties of the customers and vehicles, including their distribution throughout the network; and so on. A benchmark would standardize these factors and provide guidance for algorithm design. Existing benchmarks for the related Dial-A-Ride problem (DARP) [15] cannot be adapted because (1) DARP does not consider road networks; (2) the customers and vehicles do not represent real ridesharing scenarios; and (3) the datasets are small compared to ridesharing instances [28, 30, 19]. Thus separate benchmarks for ridesharing are needed.

Third, cross-study evaluations of algorithms from independent studies with precise analysis of strengths and weaknesses are currently missing, difficult due to the complex nature of the ridesharing problem. For example, evaluation procedures must address how to handle real-time *match latency*; they must also measure total travel distance of the vehicles over the period of a scenario, crucial for comparing algorithm quality. In terms of techniques, the database community has focused on filtering, data structures, and heuristics to achieve algorithm speed while others have emphasized grouping and other procedures for improving quality. A standardized evaluation of these different approaches is necessary for understanding their merits, drawbacks, and applicability to real-world ridesharing problems.

To fill these gaps, we make the following contributions:
(1) We develop the Cargo system for implementing and evaluating ridesharing algorithms through online, real-time simulation. Our system provides standard components and routines to ease implementation, and offers a single evaluation protocol for ridesharing events (vehicle motion, assignment events, etc.) that can be used to objectively compare dif-

Proceedings of the VLDB Endowment, Vol. 12, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3339490.3339493

[1]Guoliang Li is the corresponding author.



$$\mathscr{E} = \left\{ \begin{array}{c|ccccccc} & a & b & c & d & e & f & g \\ \hline a & & 1 & & & & & \\ b & 1 & & 4 & & 2 & & \\ c & & 4 & & 2 & & 5 & \\ d & & & 2 & & & & \\ e & & 2 & & & & 6 & \\ f & & & 5 & & 6 & & 2 \\ g & & & & & & 2 & \end{array} \right\}$$

	origin	dest.	early	late	cap.	load	speed
Vehicle k	a	g	0	30	2		1
Customer r_1	d	e	1	18		1	
Customer r_2	d	f	2	25		1	

Figure 1: *Ridesharing example.*

ferent algorithms. Our system is open source, available to practitioners for conducting online experiments [1].

(2) We present a set of benchmark instances for the standardized evaluation of algorithms. Our benchmark is designed based on real road networks with different features (*e.g.* sizes and geometries) and on real ridesharing customers and vehicles. The benchmark allows us to manipulate properties of the problem to test different situations.

(3) We present an experimental comparison of recent representative algorithms using our benchmark and system, and we report comprehensive findings from our experimental evaluation. We provide new insights on the strengths and weaknesses of existing algorithms that can guide practitioners to select appropriate algorithms for various scenarios.

The rest of this paper is organized as follows. Section 2 defines the ridesharing problem. Section 3 describes the algorithms we evaluated. Section 4 describes the Cargo system. Section 5 describes the benchmarks used in this paper. Section 6 presents experimental results and our analysis. Finally, Section 7 concludes our work.

2. BACKGROUND

2.1 Preliminaries

Table 1 summarizes our notation.

Road Network. Let a directed graph $\mathscr{G}(\mathscr{V},\mathscr{E})$ represent a road network, with nodes in $\mathscr{V}$ representing intersections and edges in $\mathscr{E}$ representing streets. Each edge $(u,v) \in \mathscr{E}$ has a weight w_{uv} (*e.g.* travel distance along the street). A route is another name for a path through $\mathscr{G}$, and its cost is the sum of the weights of all the edges in the path.

Vehicles and Customers. Let vehicles and customers be represented by origin-destination pairs. For simplification, assume that these are bound to nodes in $\mathscr{V}$. If an origin or destination is not exactly at a node, it can be mapped to the nearest node using existing methods [42].

Each vehicle k begins a *transportation service* at its origin k_o and ends the service at its destination k_d. It moves by following a route from k_o to k_d with a certain speed. Its earliest possible departure time from k_o (early time) and its latest acceptable arrival time at k_d (late time) form the vehicle's *time window* $[e_k, l_k]$, where $e_k < l_k$.

Each customer r begins at its origin r_o and requests to be transported to its destination r_d by traveling with a vehicle that is in service. The assigned vehicle will move to r_o to pick up the customer, then to r_d to drop off the customer, thereby satisfying the customer's travel request. Each customer also has a time window $[e_r, l_r]$. The early time e_r gives the earliest possible time a vehicle can pick up customer r, and the late time l_r gives the latest drop off time that the customer can accept. Note that time windows on the vehicles and customers generalize "maximum detour" and similar service guarantees encountered in the real world.

Table 1: *Notation used in this paper.*

Notation	Description
n, m	Number of customers, vehicles
$r \in N, k \in M$	Set of customers, vehicles
r_o, r_d	Customer origin, destination
k_o, k_d	Vehicle origin, destination
e_k, e_r	Early time for vehicle k, customer r
l_k, l_r	Late time for vehicle k, customer r
Q_k, q_r	Capacity of vehicle k, load on customer r
k_{gps}	Current location of vehicle k
$\mathtt{ud}(u,v)$	Euclidean distance from u to v
c_{uv}	Shortest-path cost from u to v
$S = \{u_1, \ldots, u_s\}$	Schedule with length s
$C(S,k)$	Cost of schedule S taken by vehicle k

Each vehicle k has an initial capacity $Q_k > 0$, and each customer r is associated with a load $q_r > 0$. When a vehicle picks up customer r, its capacity decreases by q_r; when the vehicle drops off r, its capacity increases by q_r.

Example 1. We use Figure 1 as a running example. The edges $\mathscr{E}$ of the road network are given in the adjacency matrix; the nodes are $\mathscr{V} = \{a,b,c,d,e,f,g\}$. Vehicle k has origin at node a, destination at node g, and moves 1 unit of distance per unit of time. Customer r_1 aims to travel from d to e, and customer r_2 aims to travel from d to f. The bottom table shows time windows, capacities, and loads.

2.2 Ridesharing Problem (RSP)

Given vehicles and customers on a road network, the goal of the Ridesharing Problem (RSP) is to find the minimum-cost set of vehicle routes that can serve all customers, subject to the following constraints:

- *Precedence.* A vehicle must visit r_o first, and then r_d, in order to serve customer r.
- *Time Windows.* Each vehicle k must depart from k_o and arrive at k_d within the time window $[e_k, l_k]$. For each customer r that it serves, it must serve the customer within the time window $[e_r, l_r]$.
- *Capacities.* Each vehicle's capacity must always be positive or zero.

Example 2. In the example, the solution is to route vehicle k through the node sequence $\{a,b,c,d,c,b,e,f,g\}$, serving both customers. This route has a cost of 23. If k, r_1, and r_2 traveled individually (no ridesharing), the sum of their individual costs is 27.

Online Assignment. In the online problem, vehicles and customers are not known beforehand, but revealed throughout the day at their early times. Each revealed customer must be assigned to a vehicle that is in service, and then the route for that vehicle must be adjusted in order for it to serve the customer. The objective is to minimize the total route cost across all the vehicles at the end of the day.

As vehicles are given new assignments, their routes can change. For each vehicle, its final route cost after it has ended service is used to compute the total route cost.

Optimization Goals. As not every customer can be feasibly served in the real world due to tight time windows, some studies relax the requirement to serve all customers, then add a penalty on the total route length for the unmatched customers [38, 6, 22, 34]. Others formulate the online problem as a series of offline instances, and then aim to optimize each of the instances [27, 20, 14, 41], while others simplify the problem to one-to-one matching where vehicles have only one capacity [36, 29, 9, 5]. Other studies use profit maximization as the objective [41]. Some studies do not aim at global optimization, instead offering customers choices based on price and time [11, 12, 9].

Vehicle Schedule. Given a customer request, we first need to find a valid vehicle for the customer, which is one where there exists at least one route for the vehicle to serve the customer while satisfying all the constraints. We then adjust the route of the vehicle.

The minimum-cost route for a vehicle can be expressed using the vehicle's schedule. Let schedule $S = \{u_1, \dots, u_s\}$ be an ordered sequence of s unique customer origins and destinations u_i, where $1 \leq i \leq s$ gives the position of the origin or destination in the schedule. Each arrangement of the elements produces a different schedule. By following a particular schedule, vehicle k visits each of the origins and destinations in given order while it travels from k_o to k_d. The shortest route through this schedule is formed by the shortest paths through each of the schedule's elements. Let c_{uv} be the shortest path cost from node u to node v. Then,

$$C(S,k) = c_{k_o,u_1} + \sum_{i}^{s-1} c_{u_i,u_{i+1}} + c_{u_s,k_d} \tag{1}$$

expresses the cost of schedule S based on the shortest route for vehicle k. The minimum-cost route for k thus corresponds to the least-cost permutation of S using Equation 1.

Example 3. The best schedule for vehicle k in the example is $\{a, d, d, e, f, g\}$, with cost $c_{ad} + c_{dd} + c_{de} + c_{ef} + c_{fg} = 23$.

Problem Complexity. If all vehicles have their origin and destination at a single common depot, then the RSP is equivalent to the DARP. The DARP is known to be NP-hard because it generalizes the Traveling Salesman Problem [15]. Having multiple vehicle origins and destinations does not lose generality, thus the RSP is NP-hard.

Relation to Ridehailing. We make the following distinction. Ridehailing vehicles, as opposed to ridesharing vehicles, have no late windows and no destinations of their own. These "taxis" can be modeled in the RSP by giving them infinite late windows and some imaginary destination. Ridesharing is clearly more general. In preliminary tests, we found that taxis can double the number of matches over ridesharing vehicles because they can continually serve requests. Hence, all our benchmarks in Section 5 use taxis.

2.3 Related Work

DARP Benchmarks. Many benchmarks exist for the DARP [28, 30, 19], but they cannot be adapted to the RSP as discussed in Section 1. First, the RSP and DARP operate on different graphs. In the DARP, the graph is complete and for n customers there are totally $2n + 2$ nodes. These nodes represent all customer pickups and dropoffs, and include two extra nodes for an "origin depot" and "destination depot". A weighted edge exists for each node pair. All weights are given and can represent distance, time, or some other metric. Each vehicle route (path through the graph) must begin from the origin depot and end at the destination depot. In the RSP, the graph is not complete. Each node maps to a physical location on a road network. A path through this graph corresponds to a physical route through the road network. The number of nodes depends on the size of the road network and the mapping method. An edge exists between a pair of nodes only if there is a physical road segment that connects them without passing through other nodes. Edge weights represent distance and can be converted to time if the travel speed along the road segment is known.

Second, the RSP is online in practice. DARP requests are usually known in advance [15] whereas RSP requests arrive continuously throughout the day. Some DARP methods may be used on the RSP, but they must be adapted for road networks and for the online scenario. To our knowledge, two DARP methods have presently been tried on the RSP. We point them out in Section 3.

Comparison Studies. Some small-scale comparison studies exist. One study compares three RSP strategies on a simulation of the Seoul road network [22] using 600 vehicles and a customer request rate of around 1.3 requests per second. Another study [14] uses up to 500 vehicles and a rate of around 5.6 requests per second on simulations of the New York City and Chicago road networks. Our evaluation in Section 6 adds to existing comparison studies by evaluating more algorithms and at real-world scale.

Experimental Platforms. Some platforms exist for testing logistics and other road network problems. In [4], a simulator is presented that can model millions of agents over a large range of mobility decisions. Similarly, in [23] a traffic simulator is presented to investigate vehicle routing choices. In [39], a specialized platform is designed to test algorithms for pickup-and-delivery problems, and in [8], a platform for generating moving objects on a road network is introduced. With modification, some of these platforms may support evaluating RSP algorithms. Our platform is designed specifically for ridesharing by including components and routines common to ridesharing algorithms.

3. RIDESHARING ALGORITHMS

In this section, we discuss an exact algorithm for the offline problem and several online algorithms. We present our evaluations on all algorithms in Section 6 using the system and benchmarks described later.

3.1 Branch-and-Bound

Branch-and-Bound (BB) [26] is a general technique for solving mixed-integer linear programs. It has been used for the DARP [15] and can be directly applied to the offline RSP. The RSP can be formalized as an integer minimization program in order to use BB.

Formalization. An integer-based labeling scheme is used to formulate the program. Let m be the number of vehicles and n the number of customers. Now the vehicles can be labeled from 1 to m, and the customers from 1 to n, to form two sets of integer labels $\mathbb{M}$ and $\mathbb{N}$, respectively. Using these sets, the origins and destinations of the customers and vehicles can be labeled. For each vehicle with label k where $k \in \mathbb{M}$, its origin can also be labeled k, and its destination can be labeled $m + k$. All the vehicle origin labels thus form the set $M_o = \{1 \dots m\}$, and all the vehicle destination labels form the set $M_d = \{m + 1 \dots 2m\}$. In a similar manner, for each customer with label r where $r \in \mathbb{N}$, its origin and destination can be labeled to produce the set of all customer origin labels $N_o = \{2m + 1 \dots 2m + n\}$ and the set of all customer destination labels $N_d = \{2m + n + 1 \dots 2m + 2n\}$. Now, the RSP is formalized as the optimization problem

$$\text{Minimize} \sum_{k \in \mathbb{M}} \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij}^k, i \neq j \tag{2}$$

where $V = N_o \cup N_d \cup \{k, k + m\}$ is the union of all possible origins and destinations that vehicle k can visit, c_{ij} is the shortest-path cost from the stops represented by i and j, and x_{ij}^k is a binary decision variable equal to 1 if k travels from stop i directly to stop j in its route, and 0 otherwise.

Table 2: ***Ridesharing algorithms. Complexity is written in terms of the customer insertion cost c_{in}. For simple insertion, $c_{in} = O(s^3 \times c_{sp})$ where s is schedule length and c_{sp} is the cost of finding one shortest path. Areas: OR=Operations Research; DB=Database; TR=Transportation; AI=Artificial Intelligence.***

Algorithm	Area	Type	Method	Complexity (n customers, m vehicles)
Branch-and-Bound (BB) [15]	OR	-	Exact	$O(2^{\left(m\times(2n+2)^2\right)})$
Nearest Neighbor (NN) [22]	TR	Search	Heuristic	$O(n(m \times c_{in} + m\log m))$
Greedy Insertion (GR) [21, 27, 22]	DB, TR	Search	Heuristic	$O(n(m \times c_{in}))$
Greedy Kinetic Tree (KT) [20]	DB	Search	Heuristic	$O(n(ms! \times c_{sp}))$
Bilateral Arrangement (BA) [14]	DB	Search	Heuristic	$O(n(m \times c_{in}))$
Simulated Annealing (SA) [22]	TR	Join	Metaheur.	$O(n(PTm \times c_{in}))$, P = perturbations, T = temperatures
GRASP (GP) [34]	AI	Join	Metaheur.	$O(g(mn^2 + h) \times c_{in})$, g = initial solutions, h = improvement iters.
Trip-Vehicle Grouping (TG) [6]	AI	Join	Exact	$O((n^2 + mn + m\sum_{i=1}^{t}\binom{n}{i})) \times c_{in})$, T= maximum trip size

Constraints. Equation 2 is subject to constraints on the routes (Section 2.2) including that each served customer must be served by the same vehicle, and that each vehicle must begin at its origin and end at its destination. The labeling scheme allows these constraints to be generally expressible, as demonstrated in [15].

Branch-and-Bound. A search tree is used to explore solutions to the above formalization. The search tree works by listing solutions to "relaxed" problems, where the integer constraint on the x_{ij}^k variables is removed (but they are still bounded by 0 and 1). Each node in the tree represents one such solution, and different solutions are obtained by fixing certain variables to be 0 or 1 for specific values of i, j and k. Each node has a cost equal to the value of the solution given by Equation 2. The tree is searched as it is constructed.

To initialize the tree, the root node is constructed by solving the relaxed problem, where $0 \leq x_{ij}^k \leq 1$ for all i, j, and k, using simplex [32] or other methods. The cost of this solution establishes the lower bound on the minimum cost by nature of being optimal. If this solution is already integer (all variables are integer), it is the optimal solution to the original problem and we are done. Otherwise, the rest of the tree is iteratively searched and constructed as follows.

(1) Branch. Create two child nodes for every node that represents a non-integer solution (in the first iteration, there is only the root node). Each child takes the same relaxed problem as its parent, except randomly select specific values for i, j and k and set $x_{ij}^k := 0$ for one child, and $x_{ij}^k := 1$ for the other. Now both child nodes represent two new relaxed problems, each with one less binary variable.

(2) Bound. Solve each new relaxed problem to obtain new solutions. For each new solution:

- If the solution is non-integer and there is no current upper bound or its cost is below the bound, take its cost to be the new upper bound, and then follow (1) to branch the node; otherwise prune the node.
- If the solution is integer, then if there is no current *incumbent* solution, or if its cost beats the current incumbent, it becomes the new incumbent and no longer needs to be branched; otherwise, prune the node.

When no more nodes are left to be branched, then the final incumbent is the optimal integer solution.

The worst-case complexity of Branch-and-Bound is exponential. In this case, all nodes are feasible and no nodes are pruned. There are $m(2n+2)^2$ binary variables (number of vehicles multiplied number of i, j pairs), each with two possible values; hence the complexity is $O(2^{(m(2n+2)^2)})$. The exponential complexity means only small instances are solvable within reasonable time.

3.2 Online Algorithms

Algorithms for the online RSP can be classified as search-based or join-based. Both kinds depend on candidates filtering and customer insertion. Candidates filtering prunes vehicles that cannot feasibly serve a given customer while customer insertion best positions a customer's stops into an existing vehicle's schedule. We discuss these first, then discuss search and join-based algorithms.

Candidates Filtering. For vehicle k traveling at speed ν_k, d_{max}^k is the maximum distance that this vehicle can be from r_o for it to feasibly serve customer r because of the late time l_r. Let t be the current time. The duration $t_{max} = l_r - t$ gives the maximum that the customer can be served within. Now let t_{min} be the direct travel time between r_o and r_d at the vehicle's speed. From the time difference, we can compute $d_{max}^k = \nu_k\,(t_{max} - t_{min})$. If $d_{max}^k < 0$, then vehicle k cannot feasibly serve this customer. Using d_{max}^k, the candidates filter can be defined as a predicate

$$P_{ud}(k) := \left(\mathtt{ud}(k_{gps}, r_o) \leq d_{max}^k\right),$$

where $\mathtt{ud}(u, v)$ gives Euclidean distance between u and v and k_{gps} gives the current location of vehicle k. Thus given the set of all vehicles M, the candidates set K_{cands} is given by

$$K_{cands} = \{k \in M \mid P_{ud}(k) \text{ is true}\}.$$

More complex filters try to return those *most likely* to be the match. For example, [27] introduces a specialized grid index to support filtering on both r_d and r_o, [37] develops a three-tier cluster-based index to support approximate filtering using stored distances, and [38] uses pre-sorted distances to prune vehicles above a certain distance bound.

Customer Insertion. For customer r and vehicle k, a minimum-cost *augmented* schedule S^+ can be formed by inserting r_o and r_d into k's current schedule S, expressed as

$$S^+ = \underset{S' \in \mathbb{P}(S \cup \{r_o, r_d\})}{\arg\min} \; C(S', k) \qquad (3)$$

where $\mathbb{P}(S \cup \{r_o, r_d\})$ lists all permutations of the new schedule that obey the precedence constraint. Finding S^+ is equivalent to solving the Traveling Salesman Problem with Precedence Constraints [31], known to be NP-hard.

An exhaustive insertion method lists every $(s!/2^s)$ valid schedule permutation for the schedule with length s, computing the cost of each permutation and returning the one with least cost. Each $C(S', k)$ requires finding $s-1$ shortest-paths (Equation 1), so the total complexity is $O(s!/2^s \times (s-1)\,c_{sp}) = O(s! \times c_{sp})$ for c_{sp} cost of one shortest path. This method can find the *exact* least-cost schedule because it reorders existing stops, but it is expensive when s becomes large and there are many augmented schedules to compute.

Alternatively, a simple insertion method achieves polynomial time complexity by fixing the existing stops instead of allowing them to reorder, thereby computing only $O(s^2)$ permutations. The total complexity for simple insertion is thus $O(s^2 \times (s-1)\,c_{sp}) = O(s^3 \times c_{sp})$.

To speed up insertion, time window and capacity constraints can be used to prune infeasible permutations, reducing the size of $\mathbb{P}$. In [20], a kinetic tree is used to speed up exhaustive insertion, while in [21, 27, 33, 38], methods for simple insertion are developed. These methods guarantee S^+ will be feasible but not necessarily minimum cost.

3.2.1 Search Algorithms

Search-based algorithms [38, 14, 27, 20] (Algorithm 1) use *vehicle selection* to assign customers sequentially as they come online. The goal is to match the best vehicle k^* with a given customer r, depending on the decision conditions. The procedure usually couples customer insertion with vehicle search because most search algorithms depend on computing augmented schedules. After k^* is found, its route is then adjusted to serve the customer. Now, let

$$P_{mat}(k) := (\text{true if } \textit{decision conditions}) \tag{4}$$

be a predicate defining conditions for a match. For a given set of vehicle candidates K_{cands}, vehicle selection returns $k^* \in K_{cands}$ given by the relational expression

$$k^* = \sigma_{P_{mat}}(K_{cands}).$$

The challenge is how to design the decision conditions in Equation 4 towards the RSP's cost minimization objective while evaluating P_{mat} online, with no information about future vehicles. We are aware of four heuristic approaches. We first discuss a distance-based method, then introduce two cost-based greedy algorithms, and finally discuss an approach that uses a replace heuristic to try to improve quality.

Nearest Neighbor. Nearest Neighbor (NN) [22] uses Euclidean distance (`ud`) as a heuristic. The predicates

$$P_{near}(k) := \left(\text{true if } k = \underset{k \in K_{cands}}{\arg\min}\ \mathtt{ud}(k_{gps}, r_o)\right),$$
$$P_{mat}(k) := (P_{near}(k) \wedge (k \text{ is valid})),$$

express the decision conditions, where predicate P_{near} is true if k is the nearest vehicle, and P_{mat} is true only if k is also valid for the customer, otherwise the problem constraints will not be satisfied. A priority queue can be used to rank each vehicle k by `ud` from its current location k_{gps} to customer origin r_o. The complexity is $O(m \log m)$ for m vehicles as there are m number of $O(1)$ distance computations, and queue insertion is $O(\log m)$. Vehicles can then be accessed from nearest-first and checked if valid, and the first valid vehicle can be returned as the match.

Vehicle k is valid if the route through the augmented schedule after inserting r_o and r_d satisfies the problem constraints. In the worst case, no vehicles are valid. Testing one vehicle requires $m \times c_{in}$ effort for the m vehicles and c_{in} complexity of the insertion method. The worst-case complexity is this plus the cost of ranking the vehicles, or $O(m \times c_{in} + m \log m)$, then multiplied n for all customers.

Greedy Insertion. Greedy Insertion (GR) [21, 27, 22] uses a more sophisticated *cost* heuristic to improve quality. The heuristic can be expressed as

$$P_{greedy}(k) := \left(\text{true if } k = \underset{k \in K_{cands}}{\arg\min}\ \Delta cost\right),$$
$$P_{mat}(k) := (P_{greedy}(k) \wedge (k \text{ is valid})),$$

where $\Delta cost = C(S^+, k) - C(S, k)$ is the *cost increase*, giving the difference between the cost of the augmented schedule S^+ and the current schedule S. Customer r can be inserted into each vehicle and $\Delta cost$ can be computed while remembering the best one. Vehicle validity can be checked at the same time because the route for k is also computed while computing $C(S^+, k)$, and this route can be checked against

```
Search Algorithm
    Given: Set of Vehicles $M$
    Input: Customer $r$
    Output: Vehicle $k^*$, Schedule $S^+$
1: $K_{cands}$ = FilterCandidates($r, M$)
2: $k^*$ = VehicleSelection($r, K_{cands}$)
3: $S^+$ = CustomerInsert($r, k^*$)
4: return $k^*, S^+$
```

Algorithm 1: *Search Algorithm.*

constraints. One customer insertion is needed to find S^+, so total complexity is $O(c_{in})$ multiplied by mn for all the m vehicles and n customers. This complexity beats NN by an $(m \log m)$ term, but the bound is tighter on GR because it must always check all the vehicles in K_{cands}.

Kinetic Tree. To improve quality, Kinetic Tree (KT) [20] uses exhaustive instead of simple insertion for computing $\Delta cost$ during evaluation of P_{greedy}. To speed up insertion, a *kinetic tree* is used to remember only the valid schedules for a vehicle by pruning invalid ones from the tree. In this way, only the feasible subset of all possible schedules need to be considered. But in the worst case, this method still computes all insertion possibilities and thus has the same complexity as exhaustive insertion, $O(s! \times c_{sp})$ for one vehicle. The advantage is KT can find the *exact* least-cost feasible schedule per vehicle because it can reorder existing stops, at the expense of exponential complexity.

Bilateral Arrangement Bilateral Arrangement (BA) [14] adds a *replace* procedure to improve quality. To prepare, the algorithm batches customers, then lists the *valid* candidate vehicles per customer. After this preparation step, for each customer r in the batch, it finds the greedy vehicle k^* that makes $P_{greedy}(k*)$ true. If this vehicle is already valid, it accepts the vehicle immediately as the match. Otherwise, it tries to replace one unserved customer from the vehicle with r, accepting the replacement if the vehicle now becomes valid. The complexity is the same as GR.

By removing an existing customer, a vehicle may gain enough time and capacity to serve r. Note that as the steps are performed on the customers sequentially, vehicle schedules may change as customers are assigned. Thus a vehicle that was valid for r after the preparation step can become invalid later on, hence the validity check.

3.2.2 Join Algorithms

Join-based algorithms [34, 22, 6] execute on sets of customers, aiming to group customers by vehicles. They return sets of assignments and schedules for the vehicles. These algorithms wait for a certain period in order to batch customers into set R, then assign these customers all at once. These algorithms may achieve better matches than search algorithms because they consider multiple customers at a time. However, vehicle join is computationally more expensive. Given a set of candidates K_{cands}, vehicle join returns a set of assignments A given by the relational expression

$$A = R \bowtie_{P_{mat}} K_{cands}, \tag{5}$$

where $a \in A$ is a relational tuple (k^*, r) matching customer r to vehicle k^*. As with search algorithms, designing the decision conditions for P_{mat} is the main challenge, but the result of joining one customer $r \in R$ with one vehicle $k \in K_{cands}$ will affect joining other customers because of the feasibility constraints (*e.g.* capacity). Thus, a simple predicate applied on all customers in R cannot ensure that the results will meet the constraints, and most approaches aim to directly return results of the join (directly return A).

```
Initialize-Improve Framework
    Given: Set of Vehicle Candidates $K_{cands}$
    Input: Set of Customers $R$
    Output: Assignments $A$, Schedules $\mathcal{S}$
1: $\mathcal{S} = \{\}$ // empty schedules set
    Initialize
2: Single-solution: initialize $A$
3: Population-based: initialize set of assignments $\mathbb{A}$
    Improve
4: repeat
5:     Single-solution: refine $A$
6:     Population-based: refine all $A \in \mathbb{A}$
7: until acceptance criteria is met
8: $A^* \leftarrow$ best $A$, $\mathcal{S} \leftarrow$ schedules from $A^*$
9: return $A^*$, $\mathcal{S}$
```

Algorithm 2: ***Initialize-Improve Framework.***

```
Grouping Framework
    Given: Set of Vehicle Candidates $K_{cands}$
    Input: Set of Customers $R$
    Output: Assignments $A$, Schedules $S$
1: $\mathcal{S} = \{\}$ // empty schedules set
    Group
2: Form subsets of customers in $R$
    Assign
3: Assign vehicles in $K_{cands}$ to each subset
4: $A \leftarrow$ individual assignments, $\mathcal{S} \leftarrow$ schedules from $A$
5: return $A$, $\mathcal{S}$
```

Algorithm 3: ***Grouping Framework.***

We know of three such approaches. We first discuss two metaheuristics that take a set of initial assignments obtained from a heuristic method, then try to improve the assignments using additional procedures until acceptance criteria are met (Algorithm 2). We call this the initialize-improve framework. The first of these methods is a single-solution method. It uses one initial set of assignments, then continually improves this set. The second is population-based. It uses several initial sets of assignments, improves each one, then finally selects the best set. Other general single-solution and population-based metaheuristics for optimization problems [17] may also be possible, but we focus on the two developed specifically for the RSP in current literature.

We then discuss a grouping method. Grouping algorithms divide customers into subsets, then assign vehicles to each of the subsets. A vehicle assigned to a subset will go on to serve all the customers in that subset. Algorithm 3 shows the basic framework for these approaches. We know of two specific algorithms. In [14], a grouping-based bilateral arrangement algorithm is developed, and this is shown to outperform a grouping-based greedy algorithm. However its performance is similar to regular BA, possibly because it still uses BA as the underlying assignment method. Hence we focus on [6] that proposes an entirely different approach.

Simulated Annealing. Simulated Annealing (SA) is a single-solution metaheuristic for general optimization problems. It has been used for the DARP in [7] and for the RSP in [22]. Its defining feature is its ability to temporarily accept worse local decisions in order to escape local optima:
(1) Initialize. Assign a random valid vehicle to each $r \in R$.
(2) Improve. Perform P "perturbations" for a number of T "temperatures". For each perturbation, select a random customer, then reassign it to a different valid vehicle and use customer insertion to adjust the route. If the route costs are less than before reassignment, then adjust the old and new vehicles to keep the new assignment. Otherwise if the route costs are greater, then *keep it* with some probability proportional to the current temperature, usually e^{fT} where f is some tunable value [17].

The parameters can be tuned to balance quality with computation speed. Greater T and P may improve quality because more iterations of step 2 are performed, but at the expense of running time. Larger f may reduce quality because more worse decisions will be accepted (known as *hill-climbing*), but the search can be wider because many solutions are considered that otherwise would not be. Complexity of SA is worse than GR by a factor of PT because of the additional customer insertions from step 2.

GRASP. The Greedy Randomized Adaptive Search Procedure (GRASP) metaheuristic is also a general technique for solving optimization problems. It has been used for the DARP in [18] and for the RSP in [34]. As a population-based method, its defining feature is "adaptive" construction of diverse initial solutions in order to widely explore the solution space [17]. The technique performs the following steps for some maximum number of iterations. At each iteration,
(1) Initialize. Perform the following until all customers are assigned or no more vehicles are available.

1. Randomly select a previously not selected vehicle k. For each $r \in R$ where k is a candidate, use customer insertion to find the augmented schedule cost $C(S^+, k)$.
2. Select one customer with probability inversely proportional to its cost (known as *roulette-wheel selection* [25]) and make the assignment.
3. Recompute costs for the other customers, and go back to step 2 unless no more customers can be assigned to vehicle k due to constraints. Then return to step 1.

(2) Improve. Apply the following operations to the solution to generate three "neighboring" solutions.

- Replace a random assigned customer from some vehicle k with a random unassigned customer (similar to replace operation in BA).
- Swap an assignment from some vehicle k_1 with an assignment from a different vehicle k_2.
- Rearrange a customer origin or destination in some vehicle k's schedule to come after the next origin or destination in the sequence.

From these new solutions, choose the least-cost feasible solution and keep improving it until no improvement is possible, or a maximum number of iterations is reached.

By using multiple initial solutions, GRASP aims to uncover many local minima so the global one can ultimately be chosen. The quality depends on diversity of the initial solutions and usefulness of the improvement operations. In the worst case, all customers are feasible for every vehicle, and the complexity is proportional to mn^2 customer insertions due to the recomputations in step 3 of initialization.

Trip-Vehicle Grouping. Trip-Vehicle Grouping (TG) [6] aims to achieve high quality assignments by *optimally* assigning vehicles to shareable groups of customers called trips. The challenges are forming the trips and assigning vehicles.
(1) Group. To find trips,

1. For each pair $(r, k) \in R \times K_{cands}, r \in R, k \in K_{cands}$, if S^+ formed by inserting the customer into k's schedule is valid, then keep the pair as a trip of size 1.
2. For each pair $(r_1, r_2) \in R \times R$, if a vehicle can serve the pair, keep the pair as a trip of size 2.
3. Form trips of size T, until T equals maximum vehicle capacity, by combining trips of size 1 to form more trips of size 2, and then combining customers from

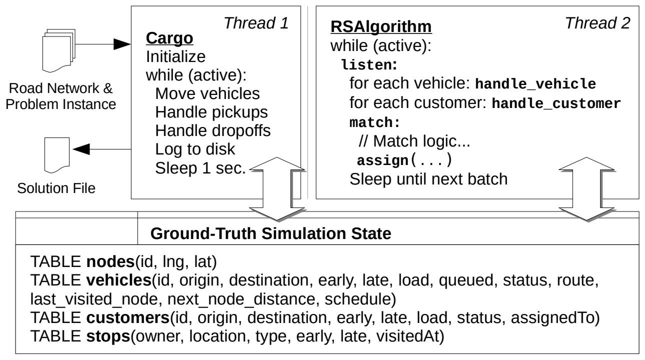

Road Network & Problem Instance
Solution File
Thread 1
Cargo
Initialize
while (active):
Move vehicles
Handle pickups
Handle dropoffs
Log to disk
Sleep 1 sec.
RSAlgorithm
Thread 2
while (active):
listen:
for each vehicle: handle_vehicle
for each customer: handle_customer
match:
// Match logic...
assign(...)
Sleep until next batch
Ground-Truth Simulation State
TABLE nodes(id, lng, lat)
TABLE vehicles(id, origin, destination, early, late, load, queued, status, route, last_visited_node, next_node_distance, schedule)
TABLE customers(id, origin, destination, early, late, load, status, assignedTo)
TABLE stops(owner, location, type, early, late, visitedAt)

Figure 2: *Cargo architecture.*

trips of size $T-1$ to form trips of size T. If a vehicle can serve all the customers in a trip, keep the trip.

(2) Assign. Solve:

$$\text{Minimize} \sum_{(i,j)\in\mathcal{E}} c_{ij}x_{ij} + \sum_{r\in R} c_{ko}y_r \tag{6}$$

to minimize sum of costs across assignments, subject to the problem constraints. Set $\mathcal{E}$ contains all possible edges between vehicles in K_{cands} and trips. An edge is formed between vehicle i and trip j if i can feasibly serve all requests in j. Cost c_{ij} is the travel distance of the best route of i in order to serve all the customers. This route can be found using exhaustive insertion or other means. Variable x_{ij} is a binary variable equal to 1 if vehicle i serves trip j, and 0 otherwise. Constant c_{ko} represents a cost penalty of not serving a request. The binary variable y_r is 1 if customer r is served, and 0 otherwise.

4. CARGO ARCHITECTURE

Cargo is a multi-threaded simulation system, shown in Figure 2, that aims to simplify the implemention of ridesharing algorithms. When simulation begins, a user-specified problem instance is loaded into an in-memory database. All customers and vehicles involved in the simulation are also loaded at this time, set to initial states.

Cargo. The main component maintains the problem instance and road network. Simulation progress is written to disk at each step for offline analysis. When simulation ends, statistics of the simulation are written into a small, uniform text file that can be used to compare algorithm performance.

Vehicle Motion. Vehicle motion is simulated by incrementing vehicle progress along their routes according to their speeds, and by updating positions (occupied nodes) when new nodes are reached. One SQL statement is used to bulk-update all vehicles. When a vehicle moves to a new location, the event is handled accordingly. Ridesharing vehicles are deactivated after they arrive at their own destination. Taxis discussed in Section 2.2 standby at their last location until given an assignment or the simulation ends.

Vehicle speed is assumed to be constant, thus travel times in the road network are static. The effects between travel times and ridesharing are not well known. Ridesharing may improve travel times by alleviating traffic congestion through sharing similar trips. But it may also *worsen* travel times by disturbing normal traffic flow and by increasing traffic as vehicles cruise for new assignments [3]. Capturing these effects in a simulation is challenging. As none of the algorithms in Section 3 are traffic-aware, we will leave studying dynamic travel times as future work.

```
RSAlgorithm Greedy Insertion (GR)
 1: function HANDLE_VEHICLE($k$):
 2:     if $k$ not in index then
 3:         Add $k$ to the index
 4:     Update index with $k$'s current position
 5: function HANDLE_CUSTOMER($r$):
 6:     $K_{cands} \leftarrow$ FILTERCANDIDATES using the index
 7:     $k^* \leftarrow$ null, $C^* \leftarrow \infty, \omega^* \leftarrow \{\}, S^* \leftarrow \{\}$
    Perform vehicle selection procedure:
 8:     for $k \in K_{cands}$ do
 9:         $S^+ \leftarrow$ CUSTOMERINSERT($r, k$)
10:         if $C(S^+, k) < C^*$ and $S^+$ is valid then
11:             $k^* \leftarrow k, C^* \leftarrow C(S^+, k), S^* \leftarrow S^+$
12:     $\omega^* \leftarrow$shortest route through $S^*$
13:     ASSIGN($r, k, \omega^*, S^*$)
```

Algorithm 4: *Greedy Insertion (GR) with index.*

Table 3: *Road network properties.*

Parameter	Manhattan	Beijing	Chengdu
Nodes	12,320	351,290	33,609
Edges	31,444	743,822	73,854
Area (km^2)	59	876	643
Total edge length (km)	1,800	17,158	5,117
Road density	30.51	19.59	7.96
Classification	dense	semi-dense	sparse

RSAlgorithm. The RSAlgorithm component represents a base ridesharing algorithm. The included functions for users to implement their own algorithms are sufficient for a wide range of ridesharing algorithms. The main functions are:

- **listen()** — Polls for active vehicles and waiting customers at a configurable interval, storing them locally.
- **handle_customer(customer)** — Executed automatically on every customer that is polled via `listen()`.
- **handle_vehicle(vehicle)** — Executed automatically on every vehicle that is polled via `listen()`.
- **match()** — Executed at the end of every `listen()` and can be used for join-based assignment.

Algorithm 4 shows an example of the greedy algorithm in [27] using RSAlgorithm. This algorithm demonstrates using a specialized index to perform candidates filtering.

Assign Mode. An assign function updates and synchronizes the database with new assignments, routes, and schedules. Synchronization is needed due to *match latency* that equals the time between when a vehicle's state is first captured to perform assignment and when the assignment is returned to the vehicle. A new assignment could be invalidated by vehicle motion during this time. For example, an assignment might instruct a vehicle to visit a node it has already passed. Cargo supports two modes. In *strict mode*, an assignment is rejected outright if it cannot be synchronized with the vehicle's current state. But in *non-strict mode*, it is accepted if the vehicle can be re-routed to accomodate the assignment. In preliminary tests, we found that the extra distance due to re-routing is more than offset by extra matches (avoiding the unmatched customers penalty). Hence we use non-strict mode in all our experiments.

Spatial Indexes. A G-tree [42] is provided for shortest-path computations, pre-built for each road network in our benchmark and loaded during initiation. A grid index is also provided for quickly filtering vehicles by Euclidean distance as explained in Section 3.2. Users can optionally bring their own spatial index, such as the ones in [27] and [37].

5. BENCHMARK

5.1 Road Networks

We used Manhattan, Beijing, and Chengdu as representative road networks in our benchmarks, listed in Table 3.

Their different sizes and densities (computed as total edge length divided by area) affect the performance of shortest-path algorithms that ridesharing algorithms rely on. Long edges have been split so that no edge is longer than 100 meters. Each network is a directed graph.

5.2 Problem Instances

Our problem instances aimed to reflect real-world scenarios as to give practical insight into algorithm performance. Toward this aim, the number of vehicles, scale of customes, their spatial distributions, and rate of customer requests were most important. We obtained datasets of physical taxi and DiDi trips and used them to generate the customers and vehicles so that spatial distributions and request rates are real. Then when customers were assigned to vehicles during evaluation, real-time locations of vehicles were simulated based on routes and speeds. Our instances are public [1].

Customers. For each road network, we extracted all taxi trips occurring in the sampling period from 6:00–6:30PM on an arbitrary day on that network. For Beijing instances, we obtained 17,467 trips, about 9.70 requests per second. For Chengdu and Manhattan instances, we obtained 8,922 and 5,033 trips, respectively, or about 4.96 and 2.80 requests per second. The requests rate can vary at other times, for example early morning hours or around sporting events. To model these changes, we removed or packed more real trips into the 30-minute period to simulate customer scales of 0.5x, 2.0x, and 4.0x for the Beijing instances. These *scale factors* multiplied the trip sampling period. For each new period, all customers in that period were packed into 30 minutes, yielding rates of roughly half, double, and quadruple the real requests rate. A small instance where we manually selected 8 customers and 2 vehicles is included to support finding the offline optimal.

Vehicles. All instances use "taxi" vehicles due to better performance as explained in Section 2.2. Initial vehicle positions were sampled from the trip datasets starting from 6:00PM and moving backwards in time until the desired number of vehicles was reached. Each sampled trip origin was used as the initial position of one vehicle. We varied the number of vehicles from 1,000–50,000 to cover real-world scale and beyond. Some studies place the real-world number of ridesharing vehicles to be 13,000 in certain cities [13]. In comparison, most studies use between 100–10,000 vehicles. Capacity varies from 1–9 to cover small passenger vehicles to minibuses. Note that Cargo allows even a 1-capacity taxi to serve an arbitrary number of customers (have long schedules) by alternating pickups and dropoffs.

Time Windows. For each customer r, its early bound was its trip request time in the raw data, relative to 6PM, and its late bound was the shortest-path time between r_o and r_d, using vehicle speed, plus a delay of 6, 12, or 24 minutes. Taxis in the real world work the entire day instead of appearing throughout the day, thus all taxis had 0 as their early bounds so they appear at the beginning of simulation.

6. EXPERIMENTS

We evaluated the algorithms in Section 3 using Cargo and our benchmarks. We added BA+, a variation of BA with two additional quality heuristics: (1) it only tries replacement if the replacing customer has a longer trip than the customer being replaced; and (2) it only accepts a replacement if the new route cost is less than before. For SA, we used SA100 and SA50 with hill-climbing factors of $f = 1.0$ and $f = 0.5$, respectively, to test usefulness of hill-climbing. For GRASP, we used GP4 and GP16 with 4 and 16 initial solutions, respectively. Offline BB was used to get the optimal solution to the small instance. Table 4 summarizes our experimental parameters.

Table 4: *Parameters and variables (defaults in bold).*

Parameter	Value
Grid dimensions	100 × 100 cells
LRU cache size	1 million shortest paths
Simulation duration	**30 min**
Road network	Manhattan, **Beijing**, Chengdu
Vehicle speed	10 meters per second
Vehicle capacity	1, **3**, 6, 9
# of vehicles	1k, **5k**, 10k, ..., 50k
Cust. scale factors	0.5x, **1.0x**, 2.0x, 4.0x
Delay tolerance	**6 min**, 12 min, 24 min

Implementation Details.

- Vehicle speed was 10 meters per second (36 kph).
- A 100 × 100 grid index was used to filter vehicles, as explained in Section 4. Each cell was roughly 300 sq. meters.
- For shortest-paths, we used a G-tree with fanout 4.
- A least-recently used (LRU) cache with 1 million elements was used to avoid repeated shortest-path calculations, as in [20]. All algorithms benefited from the cache.
- To mimic a real service guarantee, a *matching period* was used. Customers not matched within 60 seconds of their early bound were not tried again.
- Vehicles with schedules that were larger than 10 elements were pruned for all algorithms. Much time was being spent on these large-schedule vehicles. By doing this pruning, the speed of all algorithms improved dramatically.
- All algorithms except KT and TG used simple insertion; KT used kinetic trees while TG used exhaustive insertion for arranging trips in a group.
- For SA100 and SA50 we used $T = \{5, 4...1\}$ and $P = 15,000$, found to be a good balance of quality and speed.
- For TG, the GNU Linear Programming Kit (GLPK)[1] was used. We allowed top 30 customers per vehicle in step (1) and parallelized all steps as done in [6].

Runtime. Two runtime modes were used. In *dynamic* mode, simulation is real-time (one simulated second equals one real second). In *static* mode, Cargo waits until an algorithm completes before stepping to the next time instance. To be consistent, all algorithms executed with the same 30-second batch time. Specifically, search algorithms were not executed continuously, but waited until the batch time. Then, they executed sequentially on the batched customers.

Timeouts. A 30-second timeout was necessary for most algorithms to achieve real-time in dynamic mode. It was applied per customer for search algorithms and per batch for join algorithms. For TG, step (1) was given 15 seconds and the remaining time was allocated to the remaining steps.

Setting. All algorithms were implemented in C++11 and compiled with the `-O3` flag. Experiments were performed on a 2.2 GHz 64-bit Intel Xeon E5-2630 CPU server machine running the Ubuntu 16.04 operating system. Most algorithms used between 1–14 GB RAM depending on the benchmark. Six threads were used for TG.

Metrics. We measured three metrics:

- *Average customer handling time*, in real milliseconds.
- *Service rate*, the ratio of served customers to total customers (a rate of 1 indicates all customers were served).

[1]https://www.gnu.org/software/glpk

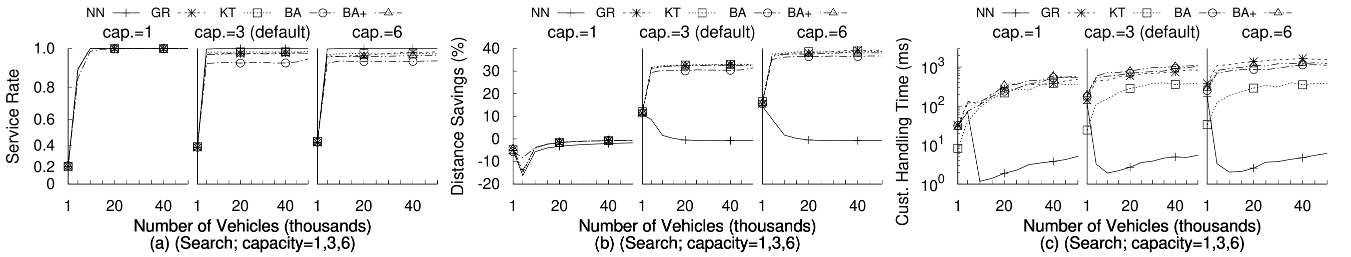


Figure 3: ***Effect of capacity on various metrics. Search algorithms, static runtime.***

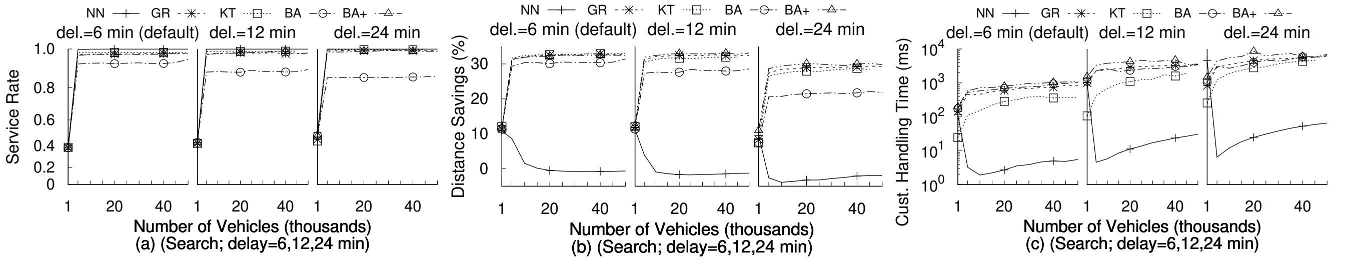


Figure 4: ***Effect of delay tolerance on various metrics. Search algorithms, static runtime.***

- *Distance savings (%)*, computed as $1 - (z/z_0)$, where

$$z = \sum_{k \in M} (\text{distance traveled by } k) + \sum_{r \in N_{ko}} c_{r_o r_d}, \quad (7)$$

$$\text{and } z_0 = \sum_{r \in N} c_{r_o r_d}. \quad (8)$$

Quantity z represents the distance achieved by an algorithm. It includes the distance traveled by all vehicles M and penalty distances for unmatched customers N_{ko}, as explained in Section 2.2. The penalty distance for customer r is the shortest-path length $c_{r_o r_d}$ from r_o to r_d. The penalty represents the real-world case where an unmatched customer might elect to take a private vehicle instead of ridesharing. Quantity z_0 is the sum of all penalties, representing the cost of no ridesharing. We call it the *base cost*. With ridesharing, z can be less than z_0 because similar trips can be combined (see Example 2). If $z > z_0$, then savings is negative, indicating surplus distance.

6.1 Static Runtime Evaluations

We first evaluated candidates filtering and customer insertion. Then static mode was used to evaluate the best performance of the algorithms. We varied the number of vehicles because it determines size of K_{cands} that can affect performance of all algorithms. We also varied vehicle capacity to evaluate different capacities used by practitioners. We omit figures for 9-capacity as no major differences were observed compared to 6-capacity. For join algorithms we varied the customer scale factor because it changes the number of customers per batch and can affect performance.

6.1.1 Filtering and Insertion

As Table 5 shows, our chosen grid size of 100 × 100 cells achieves the best time at good strength. Strength is computed here as $1 - (m/M)$ where m is the number of candidates returned by the filter, and M is the total number of vehicles. Table 5 shows the number of candidates returned by the filter was roughly linear with the filter distance d^k_{max}. Table 6 shows that customer insertion was generally an order of magnitude slower than filtering. The time grew quadratically with schedule length because permutations increased, and to perform insertion, each permutation requires multiple shortest-path computations.

Table 5: ***Filtering ($d^k_{max} = 1.8$km) by various grid size.***

Grid Cells (n^2)	$n = 1$	10	100	1,000	10,000
Avg. filter time (ms)	0.15	0.02	0.01	0.57	4.69
Strength	0%	81%	94%	96%	96%

Table 6: ***Filtering by various Euclidean distance d^k_{max}.***

d^k_{max} (meters)	225	450	900	1800	3600	7200	14400
Strength	99%	99%	98%	94%	84%	55%	8%
# of cands.	15	36	108	279	804	2241	4621

Table 7: ***Insertion by various schedule lengths.***

Sched. length	2	4	6	8	10
Avg. insertion time (ms)	0.42	1.78	3.94	6.40	9.68
# of permutations	1	6	15	28	45

6.1.2 Evaluation of Search Algorithms

Effect of Number of Vehicles. More vehicles caused handling times to increase at all capacities, but had no effect on service rate or distance savings beyond 5,000 vehicles because search algorithms could already match all of the customers at this level (Figure 3). Handling time increased due to more vehicles in K_{cands} to evaluate.

Effect of Vehicle Capacity. At 3-capacity, all except NN could achieve at least 30% distance savings (roughly 27,000 km) (Figure 3b). The savings are indistinguishable due to all algorithms using the same greedy cost heuristic. Handling time was about one magnitude higher compared to 1-capacity, with a smaller effect for KT, while NN was unchanged (Figure 3c). At 3-capacity, schedules became longer, making customer insertion harder. But NN does not rely on customer insertion and KT uses an index to perform insertion. Notably, NN was faster than all algorithms by two orders of magnitude. Service rate was generally unaffected as 1-capacity was enough to achieve high rates.

Effect of Time Window. Wider time windows had a small effect on service rate and distance except for BA (Figure 4). With wider windows, BA accepted more replacements but could not reassign the displaced customers. As more schedules became feasible, schedules that allowed more travel were accepted and distance savings worsened. On the other hand, BA+ did not experience these effects due to the additional heuristics. Handling time generally increased as there were more valid candidates to evaluate.

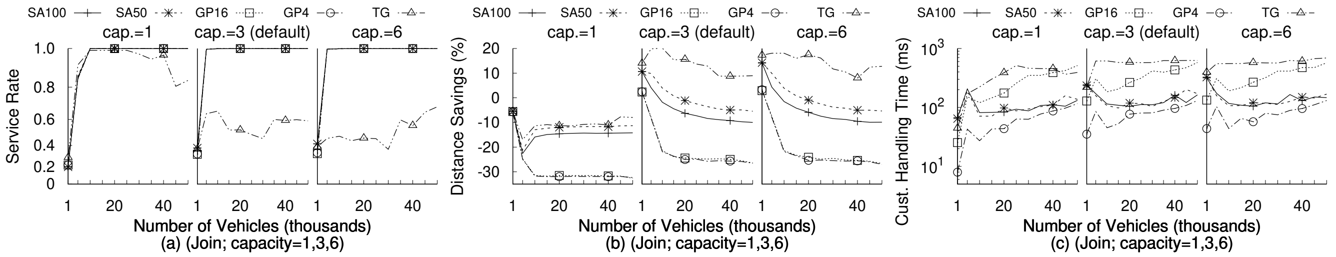


Figure 5: *Effect of capacity on various metrics. Join algorithms, static runtime.*

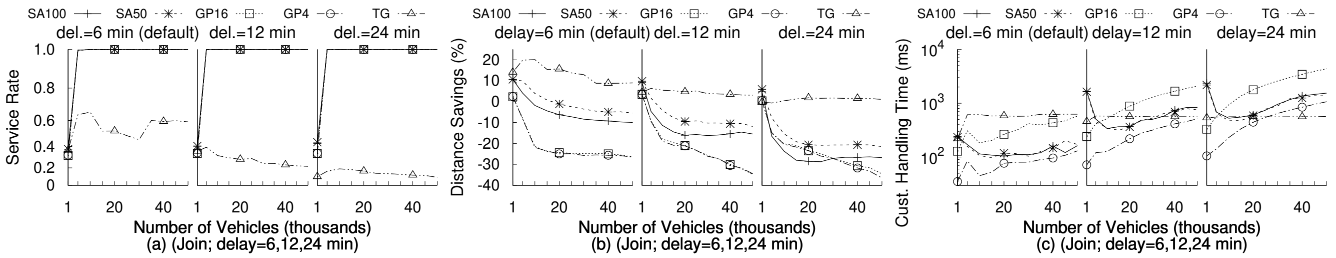


Figure 6: *Effect of delay tolerance on various metrics. Join algorithms, static runtime.*

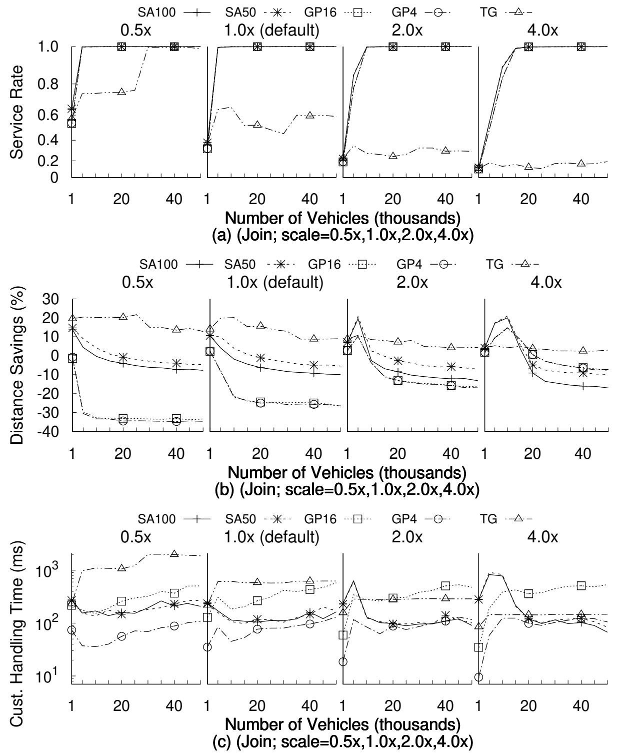


Figure 7: *Effect of scale on various metrics. Join algorithms, static runtime.*

Takeaways. Excluding NN, (1) *all search algorithms could achieve 100% service rate and more than 30% distance savings*; (2) but KT stands out due to best handling time; (3) and simple insertion is acceptable as neither BA, BA+, nor KT could improve distance savings. This result agrees with [27] that the benefit from schedule reordering is small. We did not find BA beating GR in terms of quality as in [14], possibly because the greedy algorithms differed and they used utility score while we used travel distance.

6.1.3 Evaluation of Join Algorithms

On most instances, TG took over 10 hours to run, hence a TG-specific timeout of 10 minutes per batch was added to keep times reasonable. For customer scale experiments, the requests rate ranged from 4.86 to 36.39 customers per second and the absolute number from 8,754 to 65,500 customers.

Effect of Number of Vehicles. Like search algorithms, number of vehicles had slight effect beyond 5,000 except to cause handling times to increase (Figures 5c and 6c) because all algorithms already matched near 100% at this level (Figure 5a). In all cases, SA50 used less distance than SA100 because hill-climbing knocked SA100 out of good solutions. The effect may diminish at higher P as there would be more improvement opportunities, but the handling time would increase. Also, GP4 achieved the same distances as GP16 but at half the handling time. This result suggests the improvement procedures were ineffective, as also observed in [34]. Replacement was not performed because with many vehicles, a customer was difficult to be unassigned after initialization, and swap rarely selected a valid vehicle due to many candidates in a solution where only a portion are valid for a given customer. Handling time increased for GPs and TG due to larger K_{cands}, but not for SAs as they use *random-selection* and do not evaluate all candidates. Different scales altered the effects (Figures 7a, 7b, and 7c).

Effect of Customer Scale. Scale degraded TG and had a nonlinear effect on GP and SA algorithms. Distance savings by TG decreased as scale increased (Figure 7b) because it ran against the timeout. Handling times for TG actually decreased at larger scales as it had to process more customers within the same time allotted by the timeout. For GP and SA algorithms, larger scales caused the point at which all customers could be matched to shift. For example, at 2.0x this point was at 10,000 vehicles but at 4.0x it shifted to 15,000 vehicles (Figure 7a). These algorithms only saved distance *before* this shifting “saturation” point, and handling times also peaked near the point (Figure 7c).

We try to explain these observations about the saturation point. First, regarding service rate, the number of matches for SA and GP algorithms depend on the number of vehicles. For both algorithms, the number of matches is predetermined during the initialization phases as no assignments are added in the improvement phases. If the number of vehicles is too few, SA and GP algorithms may not fully initialize the customers. The SAs loop through the customers and look for a random feasible vehicle to be a match. Without enough vehicles, some customers may not have any feasible candidates and be unmatched. The GPs loop through the vehicles and look for customers to add to each vehicle. The number of matches is thus related to the number of vehicles. At some m, all the customers can be initialized and the service rate becomes 1.

Second, two competing factors appear to affect the distance savings. Consider the case where customers are *randomly distributed* amongst m vehicles. If m is very large, then the chance that two customers share one vehicle falls. But if m is small, then not all customers may be served and the unmatched penalty will be large. At some m, sharing is maximized while the penalty is minimized and savings are greatest. As both SA and GP depend on random selection, this effect appears to a degree.

Third, we found that handling time worsened at the saturation point due to longer schedules from many shared trips, making customer insertion harder.

Effect of Vehicle Capacity. Capacity did not affect service rate or handling time except for TG. Only TG took advantage of 3-capacity to save distance over 1-capacity (Figure 7b), but with time increases of about one order (Figure 7c). No other algorithms took advantage of capacity.

At 3-capacity, TG made 60% service rate but still achieved 20% distance savings compared to 30% for search algorithms. Figure 7c shows the handling time was flat from hitting the timeout. Thus given better hardware or more processing time, TG may do better than search algorithms.

Effect of Time Window. As with search algorithms, large time windows increased handling times of all algorithms while worsening distance savings (Figure 6). As the number of feasible candidates increases, SAs should require more iterations to converge to a good solution, and the improvement procedures for GPs were still ineffective due to the previously explained effects.

Takeaways. If hardware is powerful, (1) *TG is recommended because it can reliably save travel distance* while SA and GP algorithms are sensitive to the saturation point. Otherwise, (2) SA50 is preferable over SA100 because it makes slightly better solutions due to less hill-climbing, and (3) GP4 is preferred over GP16 due to faster handling time. We could not confirm that SA uses less distance (time) than GR as in [22], possibly because that study used only 600 vehicles and maximum 1.25 customers per second, making it easier for SA to converge to a good solution.

6.2 Real-time Dynamic Evaluations

Dynamic runtime mode was used to characterize the real-world performance of the algorithms. We varied number of vehicles, capacity, and customer scale, all factors in the real world. We additionally evaluated performance on Manhattan and Chengdu road networks. We omitted BA, SA100 and GP16 based on previous results. We also omitted handling times due to the presence of the dynamic runtime timeouts. We show that algorithms mostly could not achieve near their static mode performance.

Effect of Number of Vehicles. For NN, service rate remained high at all numbers of vehicles but no distance savings could be achieved. For the other search algorithms, service rate and distance savings both fell with increasing vehicles. The decline was faster for GR and BA+ compared to KT and accelerated at larger scales (Figures 10a and 10b). From the static evaluations, handling time can increase with more vehicles because K_{cands} increases. All search algorithms (except NN) perform customer insertion on *each of the candidates*, hence take longer when there are more. But in dynamic mode, *a long handling time causes the matching rate to fall behind the requests rate.* As a result, the queue of customers waiting to be processed fills. After the 60-second matching period expires, customers drop, causing service rate to suffer. Hence the *matching rate should meet or exceed the requests rate* to ensure good service.

To elaborate, Figure 11 shows the queue size over time on an instance using default parameters. The sawtooth pattern for NN, GP4, and SA50 indicate they could match all customers within each 30-second batch, explaining their ability for high service rates. Out of the P_{greedy} algorithms, KT had the best queue behavior, explaining its ability to save up to 30% travel distance (Figure 9a). For SA50 and GP4, queue size oscillated between 300 and 600 customers because the algorithms took fully 30 seconds allowed by the timeout before returning results, in addition to the 30-second batch time. The lag caused 60 seconds worth of customers to be in the queue. For TG, the queue was nearly always full for all instances, thus it could not achieve a good service rate nor good distance savings.

Effects from the saturation point for SA50 and GP4 disappeared because the timeouts limited the processing time. Instead, GP4 experienced a "critical point" where service level dropped dramatically. For example, this point was 30,000 vehicles at 3-capacity (Figure 8a). After this point, GP4 could not initialize full solutions within the timeout due to the expensive repeated fitness evaluations, hence it returned only partial solutions.

Effect of Customer Scale. Algorithms could not meet the higher request rates at larger scales. Only NN and SA could maintain good service rates (Figure 10a). As scale increased, the critical point for GP4 shifted to lower numbers. For example, at 0.5x the critical point is beyond 50,000 vehicles, but at 2.0x it is 5,000 vehicles.

Effect of Vehicle Capacity. Capacity had little effect beyond 3 under dynamic runtime as under static runtime.

Effect of Time Window. Service rates for all algorithms except NN and SA50 dropped with wider time windows (Figure 8b). Distance savings dropped dramatically for SA50 (Figure 9b). With wider time windows, more vehicles become feasible, and the chance of randomly selecting the best vehicle per customer becomes smaller. As SA50 depends on random selection to form the initial solution, the chance of forming a good initial solution drops. As it has no time to perform many perturbations in dynamic made, it cannot improve these initial assignments and distance suffers.

Performance on Road Networks. All algorithms could make more matches and save more distance on Manhattan than on the other two networks (Figures 8c and 9c), and Beijing was the hardest. Customer scale appears to have a stronger effect than road network density. Algorithms achieved highest service rate and most distances savings on dense Manhattan, but performed only slightly worse on sparse Chengdu. Algorithms struggled the most on Beijing, but when scale was reduced to 0.5x, then algorithms could achieve better much service rates (Figure 10a).

Takeaways. Competitive analysis [35] is often used to characterize online algorithms. In lieu of theoretical competitive ratios, we ran the algorithms on a small instance against optimal BB. Table 8 shows the results (obtained after several

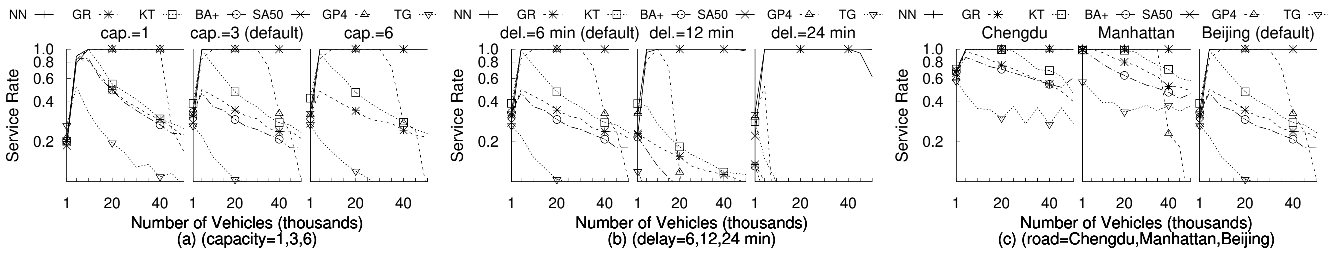


Figure 8: ***Service rates by various factors. Real-time evaluations, dynamic runtime.***

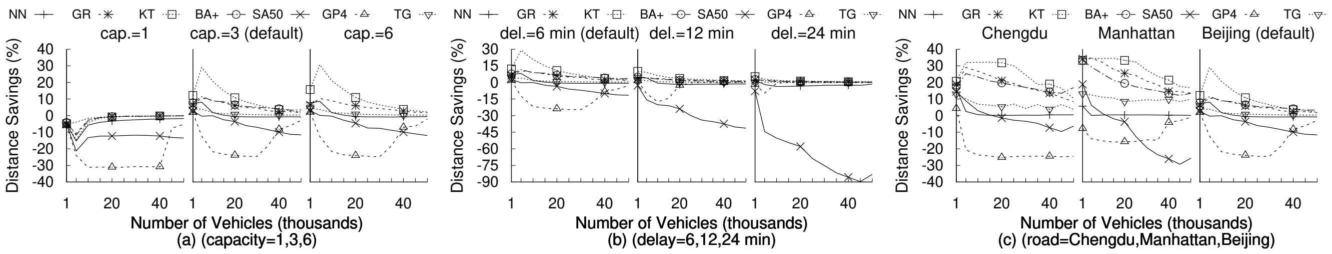


Figure 9: ***Distance savings by various factors. Real-time evaluations, dynamic runtime.***

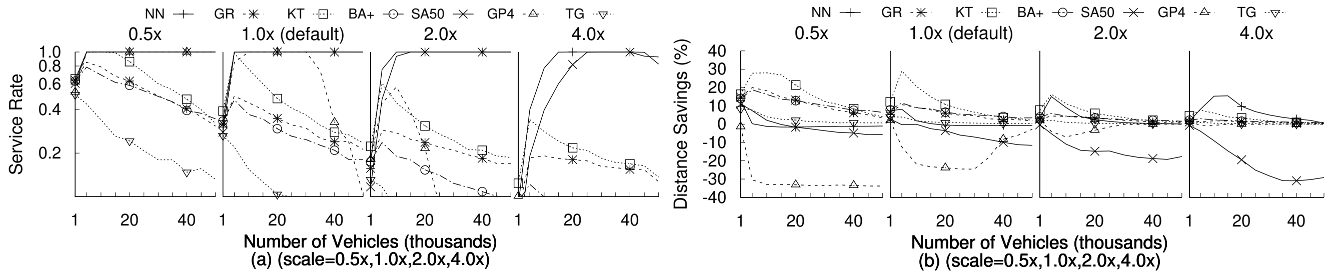


Figure 10: ***Effect of scale on service rate and distance savings. Real-time evaluations, dynamic runtime.***

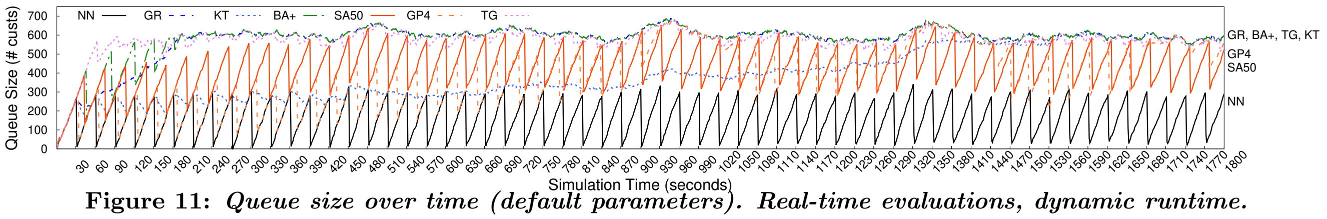


Figure 11: ***Queue size over time (default parameters). Real-time evaluations, dynamic runtime.***

Table 8: ***Competitive ratios (lower is better).***

	NN	GR	KT	BA+	SA50	GP4	TG	BBAll	BBOpt
Ratio	1.15	1.12	1.12	1.10	1.06	1.05	1.01	1.00	0.95
Time	2ms	2ms	28ms	1ms	1000ms	4ms	271ms	>1hr	>1hr

trials). The algorithm BBAll finds the optimal solution if all customers must be matched while BBOpt finds the optimal solution if this constraint is lifted (while applying penalties for unmatched customers). Notably, TG was nearly optimal because it aims to optimize groupings, but it could not take into account *future requests*. Yet in real-time ridesharing, fast algorithms are preferred. Thus, we recommend (1) NN and SA for maximizing service rate; (2) KT for maximizing distance savings; (3) and TG based on its competitiveness, if the computational burden can be overcome.

7. CONCLUSION

This work develops a system for evaluating ridesharing algorithms, proposes benchmarks to study these algorithms in online real-time cases, and performs an evaluation covering a wide range of real-world scenarios. The goal is to provide standardized testing of these algorithms, currently lacking due to the complex nature of the problem and difficulty in evaluation. We also reveal key factors explaining algorithm performance, then offer several recommendations and point the direction for practitioners and algorithm designers.

We offer the following recommendations. For practitioners: (1) If maximizing matches is important, we recommend NN or SA because they can meet the customer requests rate in most cases. (2) If maximizing distance savings is important, we recommend KT. (3) We recommend deploying 3-capacity vehicles because current algorithms yield no benefits from higher capacities. For algorithm designers: (4) Algorithms that rely on schedule costs to make assignment decisions may not meet the requests rate when there are more than a few thousand vehicles. However, indexing the vehicle schedules, for example with kinetic trees, can improve the performance. Thus future algorithms could use cheaper heuristics other than schedule cost, or use efficient schedule indexing. (5) Random-selection takes too long to converge to a good solution. Instead, selection should be guided aggressively to the "best" vehicle, either through better selection bias or better improvement procedures. (6) Unless the computational burden can be overcome, listing customer-vehicle combinations, as in TG, should be avoided. Thus, future algorithms could take advantage of these techniques: cheap heuristic, schedules index, and guided selection to outperform the state-of-the-art. Existing studies focused on static travel costs and it calls for traffic-aware methods that consider realistic traffic in online ridesharing.

Acknowledgement. This work was supported by the 973 Program of China (2015CB358700), NSF of China (61632016, 61521002, 61661166012), Huawei, and TAL education.

8. REFERENCES

[1] Cargo. https://github.com/jamjpan/Cargo.

[2] China's didi now sees... https://www.techinasia.com/china-didi-kuaidi-10-million-daily-rides.

[3] Tncs and congestion. https://www.sfcta.org/emerging-mobility/tncs-and-congestion.

[4] M. Adnan, F. C. Pereira, C. L. Azevedo, K. Basak, M. Lovric, S. Raveau, Y. Zhu, J. Ferreira, C. Zegras, and M. Ben-Akiva. Simmobility: a multi-scale integrated agent-based simulation platform. *Trans Res Board 95th Annual Meeting*, 2016.

[5] N. A. H. Agatz, A. L. Erera, M. W. Savelsbergh, and X. Wang. Dynamic ride-sharing: a simulation study in metro atlanta. *Transportation Research Part B: Methodological*, 45(9):1450–1464, 2011.

[6] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *PNAS*, 114(3):462–467, 2017.

[7] K. Braekers, A. Caris, and G. K. Janssens. Exact and meta-heuristic approach for a general heterogenous dial-a-ride problem with multiple depots. *Transportation Research Part B*, 67:166–186, 2014.

[8] T. Brinkhoff. Generating network-based moving objects. *SSDM*, 2000.

[9] B. Cao, L. Alarabi, M. F. Mokbel, and A. Basalamah. Sharek: a scalable dynamic ride sharing system. *MDM*, pages 4–13, 2015.

[10] B. Caulfield. Estimating the environmental benefits of ride-sharing: a case study of Dublin. *Transportation Research Part D: Transport and Environment*, 14(7):527–531, 2009.

[11] L. Chen, Y. Gao, Z. Liu, X. Xiao, C. S. Jensen, and Y. Zhu. PTRider: a price-and-time-aware ridesharing system. *PVLDB*, 11(12):1938–1941, 2018.

[12] L. Chen, Q. Zhong, X. Xiao, Y. Gao, P. Jin, and C. S. Jensen. Price-and-time-aware dynamic ridesharing. In *ICDE*, pages 1061–1072, 2018.

[13] P. Cheng, X. Jian, and L. Chen. An experimental evaluation of task assignment in spatial crowdsourcing. *PVLDB*, 11(11):1428–1440, 2018.

[14] P. Cheng, H. Xin, and L. Chen. Utility-aware ridesharing on road networks. *SIGMOD*, pages 1197–1210, 2017.

[15] J.-F. Cordeau. A Branch-and-Cut Algorithm for the Dial-a-Ride Problem. *Operations Research*, 54(3):573–586, 2006.

[16] A. D. Febbraro, E. Gattorna, and N. Sacco. Optimization of dynamic ridesharing systems. *Transportation Research Record: Journal of the Transportation Research Board*, pages 44–50, 2013.

[17] M. Gendreau and J.-Y. Potvin, editors. *Handbook of Metaheuristics*. Springer, 2 edition, 2010.

[18] F. Guerriero, M. E. Bruni, and F. Greco. A hybrid greedy randomized adaptive search heuristic to solve the dial-a-ride problem. *Asian-Pacific Journal of Operational Research*, 30(1):1–17, 2013.

[19] S. C. Ho, W. Y. Szeto, Y.-H. Kuo, J. M. Y. Leung, M. Petering, and T. W. H. Tou. A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B*, 111:395–421, 2018.

[20] Y. Huang, R. Jin, F. Bastani, and X. S. Wang. Large scale real-time ridesharing with service guarantee on road networks. *PVLDB*, 7(14):2017–2028, 2014.

[21] J.-J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. M. Wilson. A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. *Transportation Research Part B*, 20(3):243–257, 1986.

[22] J. Jung, R. Jayakrishnan, and J. Y. Park. Dynamic shared-taxi dispatch algorithm with hybrid-simulated annealing. *Computer-Aided Civil and Infrastructure Engineering*, (31):275–291, 2016.

[23] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker. Recent development and applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, 5(3&4):128–138, December 2012.

[24] A. Levofsky and A. Greenberg. Organized dynamic ride sharing: the potential environmental benefits and the opportunity for advancing the concept. *Transportation Research Board 2001 Annual Meeting*, 2001.

[25] A. Lipowski and D. Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 391(6):2193–2196, 2012.

[26] J. D. C. Little, K. G. Murty, D. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11(6):972–989, 1963.

[27] S. Ma and Y. Zheng. Real-time city-scale taxi ridesharing. *TKDE*, 27(7):1782–1795, 2015.

[28] M. A. Masmoudi, M. Hosny, K. Braekers, and A. Dammak. Three effective metaheuristics to solve the multi-depot multi-trip heterogeneous dial-a-ride problem. *Transport Research Part E*, (96):60–80, 2016.

[29] N. Masoud and R. Jayakrishnan. A real-time algorithm to solve the peer-to-peer ride-matching problem in a flexible ridesharing system. *Transportation Research Part B*, 106:218–236, 2017.

[30] Y. Molenbruch, K. Braekers, and A. Caris. Typology and literature review for dial-a-ride problems. *Annals of Operations Research*, 259(1-2):295–325, 2017.

[31] C. Moon, J. Kim, G. Choi, and Y. Seo. An efficient genetic algorithm for the traveling salesman problem with precedence constraints. *European Journal of Operational Research*, 140(3):606–617, 2002.

[32] K. G. Murty. *Linear Programming*, volume 1. John Wiley & Sons, Inc., 2000.

[33] M. Ota, H. Vo, C. Silva, and J. Freire. STaRS: Simulating Taxi Ride Sharing at Scale. *IEEE Transactions on Big Data*, 3(3):349–361, 2017.

[34] D. O. Santos and E. C. Xavier. Dynamic taxi and ridesharing: a framework and heuristics for the optimization problem. *IJCAI*, 2013.

[35] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[36] N. Ta, G. Li, T. Zhao, J. Feng, and H. Ma. An efficient ride-sharing framework for maximizing shared route. In *IEEE Transactions on Knowledge and Data Engineering*, pages 219–233, 2017.

[37] R. S. Thangaraj, K. Mukherjee, G. Raravi, A. Metrewar, N. Annamaneni, and K. Chattopadhyay. Xhare-a-ride: a search optimized dynamic ride sharing system with approximation guarantee. *ICDE*, pages 1117–1128, 2017.

[38] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu. A unified approach to route planning for shared mobility. *PVLDB*, 11(11):1633–1646, 2018.

[39] R. van Lon and T. Holvoet. Rinsim: a simulator for collective adaptive systems in transportation and logistics. *SASO*, pages 231–232, 2012.

[40] B. Yu, Y. Ma, M. Xue, B. Tang, B. Wang, J. Yan, and Y.-M. Wei. Environmental benefits from ridesharing: a case of beijing. *Applied Energy*, 191:141–152, 2017.

[41] L. Zheng, L. Chen, and J. Ye. Order dispatch in price-aware ridesharing. *PVLDB*, 11(8):853–864, 2018.

[42] R. Zhong, G. Li, K.-L. Tan, L. Zhou, and Z. Gong. G-tree: an efficient and scalable index for spatial search on road networks. *TKDE*, 27(8):2175–2189, 2015.