STRide: A Framework to Exploit the Structure Information of Shareability Graph in Ridesharing

Yu Chen *, Peng Cheng *, Xuemin Lin #, Lei Chen †, Wenjie Zhang #
*East China Normal University, Shanghai, China
yu.chen@stu.ecnu.edu.cn, pcheng@sei.ecnu.edu.cn
†The Hong Kong University of Science and Technology, Hong Kong, China
leichen@cse.ust.hk
#The University of New South Wales, Australia
lxue@cse.unsw.edu.au, wenjie.zhang@unsw.edu.au

***Abstract*—Ridesharing services play an essential role in modern transportation, which significantly reduces traffic congestion and exhaust pollution. In the ridesharing problem, improving the sharing rate between riders can not only save the travel cost of riders and drivers but also utilize vehicle resources more efficiently. The existing online-based and batch-based methods for the ridesharing problem lack the analysis of the sharing relationship among riders. In addition, the graph is a powerful tool to analyze the structure information between nodes. Therefore, in this paper, we proposed a framework, namely STRide, to utilize the structure information to improve the results for ridesharing problems. Specifically, we extract the sharing relationships between riders to construct a shareability graph. Then, we define a novel measurement shareability loss for vehicles to select groups of requests such that the unselected requests still have high probabilities of sharing. Our SARD algorithm can efficiently solve dynamic ridesharing problems to achieve dramatically improved results. Through extensive experiments, we demonstrate the efficiency and effectiveness of our SARD algorithm on two real datasets. Our SARD can run up to 72.68 times faster and serve up to 50% more requests than the state-of-the-art algorithms.**

I. Introduction

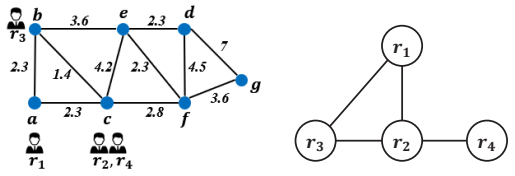
Recently, ridesharing has become a popular public transportation choice, which greatly contributes to reducing energy and relieving traffic pressure. In ridesharing, a driver can serve different riders simultaneously once riders can share parts of their trips and there are enough seats for them. The ridesharing service providers (e.g., Uber [1], Didi [2]) are constantly pursuing better service quality, such as higher service rate [3], [4], lower total travel distance [5], [6], or higher total revenue [7], [8].

For ridesharing platforms, *vehicle-request matching* and *route planning* are two critical issues to address. With a set of vehicles and requests, vehicle-request matching filters out a set of valid candidate requests for each vehicle, while route planning targets on designing a route schedule for a particular vehicle to pick up and drop off the assigned requests.

If any vehicle can serve two requests during the same trip, we call that they have a shareable relationship. Then, the shareable relationships between requests form a *shareability graph* of the requests, where each node represents a request, and each edge indicates the connected two requests can share some part of their trips. The structure information of graphs can reveal many important properties (e.g., k-core [9], [10], [11], [12], k-truss [13], [14], clique [15], [16], [17]) and contribute to specific applications (e.g., community discovery [18], [19], [20], anomaly detection [21], [22], influential analysis [23], [24], etc). However, to the best of our knowledge, no existing works have targeted the utilization of the *structure properties* of the shareability graph to guide the vehicle-request matching to improve the service quality.

TABLE I: Requests Release Detail.

request	source	destination	release time	deadline
r_1	a	d	0	14
r_2	c	f	0	11
r_3	b	e	2	10
r_4	c	g	3	9



(a) Road Network (b) Shareability Graph

Fig. 1: A Motivation Example

In this paper, we exploit the graph structures in the shareability graph of requests to achieve better results for ridesharing services. In the following, we will illustrate our motivation with Example 1.

Example 1. *In the ridesharing example shown in Figure 1(a), there are four requests $r_1 \sim r_4$ with a road network consisting of seven nodes $a \sim g$. The value close to each edge indicates the distance between the connected two nodes. The details of the requests are shown in Table I, where the release time represents the time the request submitted to the platform, and the deadline indicates the latest time to reach the destination. Suppose that the platform only allows one vehicle to service up to two requests in a trip, and it takes one unit of time to move one unit of distance in the road network. From the information given above, we have that r_1 can share with r_2 and r_3; r_2 can share with r_1 and r_3. But r_4 can only share with r_2. We can construct a shareability graph of these four requests as shown in Figure 1(b), where each edge indicates*

that the connected two requests are shareable.

Once the platform makes an allocation based on the online allocation framework [5], [6], [25] (i.e., each request will be served or discarded as soon as it arrives), it will choose requests r_1 and r_2 to share a vehicle, which leads to that requests r_3 and r_4 cannot share with other requests anymore when they arrive.

Through investigating the shareability graph, the degrees of requests $r_1 \sim r_4$ are 2, 3, 2, 1, respectively. In a shareability graph of requests, a larger degree means more potential shareable requests. If we associate higher priorities to requests having lower degrees in the shareability graph (e.g., request r_4), we can get a better dispatch result: r_4 shares with r_1, and r_2 shares with r_3.

In real platforms, vehicles and requests often arrive dynamically. Existing works on ridesharing can be summarized into two modes: the *online mode* [26], [5], [6], [27] and *batch mode* [28], [29], [30], [31]. The state-of-the-art operator for the online mode is *insertion* [27], [26], [32], [30], [25], which inserts the requests' sources and destinations into proper positions of the vehicle's route without reordering its current schedule such that the increase of the total travel cost is minimized. Batch-based methods [28], [29], [33], [34], [30] first package the incoming requests into groups, then assign a vehicle with a properly designed serving-schedule for each group of requests.

The existing approaches have their shortcomings. Insertion operator is very fast but can only achieve the local optimal schedule for a request (e.g., minimized increase of travel cost). The batch-based methods can achieve better results (e.g., higher service rates or lower travel distances) than online methods for a period of a relatively long time (e.g., one day). But their time complexities are much higher than insertion (e.g., RTV needs to do bipartite matching, which is costly [28]). Can we have a more efficient batch-based approach to have better results for a relatively long period (e.g., one day)?

In this paper, we proposed a well-tailored batch-based framework, namely STRide, to exploit the structure information of requests' shareability graph in ridesharing systems such that the achieved results are significantly improved (e.g., higher service rates and lower travel distances) under just a slight delay of response time. *The key insight of our framework is to take the lead in integrating traditional allocation algorithms with graph analyses of the shareability graph in ridesharing problems.* In STRide, we first proposed a fast shareability graph builder, which efficiently extracts shareable relationships between requests through ellipse and angle-based filtering to only maintain the good shareable relationships. To exploit the structure information of the shareability graph, we devised the *structure-aware ridesharing dispatch* (SARD) algorithm to take the cohesiveness of requests in the shareability graph (evaluated through the graph structures) into consideration and revise the priority of each request through analyzing its shareability. SARD adopts a two-phase "proposal-acceptance" strategy, which avoids enumerating invalid groups for any specific vehicle. When we design a schedule for a vehicle to serve a set of riders, we adjust the insertion order of riders' requests based on the requests' shareability in the shareability graph, such that the maintained schedule for each vehicle is close to the optimal schedule with the same space and time complexity of linear insertion. Thus, the batch-based method can be more efficient in large-scale ridesharing problems.

TABLE II: Symbols and Descriptions.

Symbol	Description
R	a set of m time-constrained rider requests
r_i	ride request r_i of rider i
s_i	the source location of ride request r_i
e_i	the destination location of ride request r_i
c	the capacity constraint of a vehicle
G_i	a group of rider requests where $\|G_i\| \leq c$
$\mathcal{P}$	a batch of requests in a time

To summarize, we make the following contributions:

- We introduce our STRide framework, which handles incoming requests in batch mode and adjusts the matching priority of requests in each batch in Section II-B.
- We propose a graph structure called *shareability graph* for intuitively analyzing the sharing relationships among requests, and designed an efficient algorithm to generate the shareability graph in each batch in Section III.
- We devise a heuristic algorithm, namely *SARD*, for priority scheduling in each batch under the guidance of the shareability graph with theoretical analyses in Sections IV.
- We conduct extensive experiments on two real datasets to show the efficiency and effectiveness of our proposed STRide framework in Section V.

II. Problem Definition

We use a graph $\langle V, E\rangle$ to indicate the road network, where V is a set of vertices and E is a set of edges between vertices. Each edge (u, v) is associated with a weight $cost(u, v)$ to represent the travel cost from u to v. In this paper, travel cost means the minimum travel time cost from u to v.

A. Definitions

Definition 1 (Request)**.** Let $r_i = \langle s_i, e_i, n_i, t_i, d_i\rangle$ denote a ridesharing request with n_i riders from source s_i to destination e_i, which is released at time t_i and requires reaching the destination before the delivery deadline d_i.

In online scenarios, each vehicle may be assigned with a certain number of requests. Therefore, we also need to plan a feasible schedule for each vehicle with allocated requests, which forms by the sequence of pickup and drop off locations as definition 2.

Definition 2 (Schedule)**.** Given a vehicle v_j with m allocated requests $R_j = \{r_1, \ldots, r_m\}$. Let $S_j = \langle o_1, \ldots, o_{2m}\rangle$ define a schedule for v_j, where the location $o_x \in S$ is the source location or destination of a request $r_i \in R_j$.

We call the location o_i in schedule as a *waypoint*. A route S is *valid* if and only if it satisfies the following three constraints:

- **Order Constraint**. For any $r_i \in R_j$, the source location s_i appears before the destination e_i in the route S_j;

- **Capacity Constraint**. At any time, the total number of assigned riders must not exceed the maximum capacity c_j of vehicle v_j.
- **Deadline Constraint**. For any $o_x \in S$, the total driving time before o_x must satisfy the inequality 1.

$$\sum_{k=1}^{x} cost(o_{k-1}, o_k) \leq ddl(o_k). \quad (1)$$

$ddl(o_k)$ is the deadline d_i when o_k is the destination of r_i, while $ddl(o_k) = d_i - cost(s_i, e_i)$ when o_k is the source of r_i. For simplicity, we denoted $cost(s_i, e_i)$ as $cost(r_i)$ in the rest of the paper.

The vehicle has a maximum allowed detour distance at each waypoint to complete the arranged schedule on time. We introduce the maximum allowed detour time as *buffer time*:

Definition 3 (Buffer Time [26], [25])**.** Given a valid vehicle schedule $S_i = \langle o_1, o_2, \ldots, o_{2m} \rangle$, let $buf(o_x)$ be the maximum detour time at the waypoint o_x without violating the deadline constraints of its consequent waypoints.

$$buf(o_x) = \min \{buf(o_{x+1}), ddl(o_{x+1}) - arrive(o_{x+1})\}, \quad (2)$$

where $arrive(o_{x+1})$ is the earliest arriving time of waypoint o_{x+1} without any detour at previous waypoints of S_i.

With the buffer time of each waypoint, we can find a convenient pickup and drop-off time for the new coming request in $O(n)$ time [25].

Based on the above definitions, *RTV-Graph* was presented in [28] to describe the shareable relationships between requests and vehicles. The proposed solution for the dynamic ridesharing problem was based on the RTV-graph and linear programming in [28], which could achieve shorter total travel distance and higher service rates. However, constructing an RTV-graph needs to check whether any two or more requests are shareable with $O(n^c)$ time, and then check whether each vehicle can serve all requests in each trip in $O(m|T|)$ time, where T is the number of total candidate trips. Therefore, it's not efficient enough to be used in a practical large-scale application. In this paper, we proposed a shareability graph for instinctively analyzing the shareable relationships to help solve the dynamic ridesharing problem defined as follows.

Definition 4 (Dynamic Ridesharing Problem)**.** Given a set, R, of n dynamically arriving requests and a set, V, of m vehicles, the dynamic ridesharing problem is to plan a feasible schedule for each vehicle $v_i \in V$ to minimize a specific utility function.

In *batched dynamic ridesharing problem* (BDRP), we need to resolve the dynamic ridesharing problem in batch mode, which handles a few incoming requests in time period T as a batch $\mathcal{P}$ and then partitions into request groups $\mathbb{G} = \{G_1, G_2, \ldots, G_z\}$, which satisfies following conditions: $\forall G_x, G_y \in \mathbb{G}, G_x \cap G_y = \emptyset$ and $\mathcal{P} = \bigcup_{a=1..z} G_a$. Then, we try to assign request groups to valid vehicles to minimize the utility function. In this work, we refer to the unified cost function UC in [25] and defined the following utility function U:

$$U(V, \mathcal{P}) = \alpha \sum_{v_i \in V} \mu(v_i, G_{v_i}) + \sum_{G_i \in G^-} p_i \quad (3)$$

$$\mu(v_i, G_{v_i}) = \sum_{x=1}^{|S_{v_i}|-1} cost(o_x, o_{x+1}), \quad (4)$$

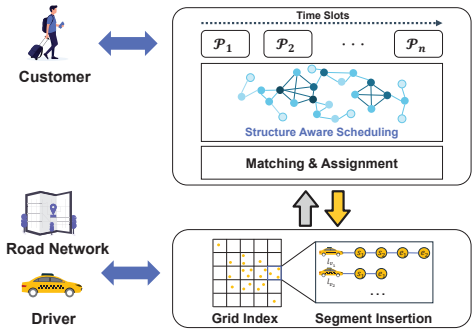


Fig. 2: Overview of System Framework *STRide*.

where S_{v_i} is the planned schedule for vehicle v_i. G^- is composed of the request groups, which has not been assigned in each batch with a penalty p_i. In this paper, we define $p_i = \beta \sum_{r \in G_i} cost(r.s, r.e)$ with a parameter β. Note that, p_i can be modified to support other optimization goals.

Table II summarizes the commonly used symbols.

Discussion of Hardness. Dynamic ridesharing problem has been proved NP-hard in many existing works [25], [26], [44], [6], thus intractable. Moreover, Tong *et al.* prove that there is no polynomial-time algorithm with a constant competitive ratio for dynamic ridesharing problem [25]. Thus, we turn to use the experimental results to show the effectiveness of our approaches. To efficiently and effectively solve BDRP, we proposed a novel batch-based framework, namely STRide, that exploits the structural information of the shareability graph of requests to improve the performance of service rates and uniform costs.

B. An Overview of STRide Framework

We first briefly introduce the essential parts of our STRide framework, as illustrated in Figure 2.

Index Structure. In ridesharing, since vehicles keep moving over time, the used index must update efficiently. Grid index is an efficient index structure that can query and update moved vehicles in constant time. In STRide, we partition the road network into $n \times n$ square cells. With the grid index, we can retrieve all available vehicles efficiently through a range query for a given location p and a specified radius r in constant time.

Schedule Maintenance. In STRide, we need to arrange the pickup and drop-off locations of the new requests to the available vehicles' schedules. There are two wildly used solutions: *kinetic tree insertion* [6] and *linear insertion* [25], [32]. Kinetic tree insertion maintains all feasible schedules for a particular vehicle in a tree structure; therefore, it always returns the optimal schedule (i.e., the shortest total traveling cost). In contrast, the linear insertion method only maintains a current optimal schedule and inserts the new points into the maintained schedule in constant time without *reordering* waypoints, which may miss the optimal schedule. However, Ma *et al.* [36] reveals that reordering waypoints almost has no change in effectiveness but needs more time and space. Therefore, we maintain the schedule for each vehicle by linear insertion.

Structure-Aware Assignment. We process the incoming requests into batches by their release timestamps and handle them in batch mode. For each batch, we utilize the shareability

graph in Section III to analyze the sharing relationship between requests. In the *Matching and Assignment* phase, we proposed a two-phase algorithm SARD in Section IV. The requests will propose to candidate vehicles in descending order of the additional travel cost (i.e., propose to vehicles needing more additional travel costs first). After that, vehicles select a group of structural-friendly requests to accept. Here, we proposed a novel measurement, namely shareability loss, to evaluate the impact of merging a group of requests into a supernode with respect to the sharing probability of the rest nodes in the shareability graph. The discarded requests will propose to other vehicles in the following proposal phase.

III. Shareability Graph

Although the requests in the same batch have similar release times, the sharing probabilities can vary dramatically. For example, a request with higher shareabilities has more opportunities to share vehicles with other requests. Thus, an intuitive motivation for optimizing the grouping process is to associate higher priority to a request with lower shareability. In this section, we proposed a *shareability graph* to manage the sharing relationships between requests with an efficient construction algorithm for it. With a shareability graph, we can further analyze the shareability of requests in each batch.

A. Shareability Graph

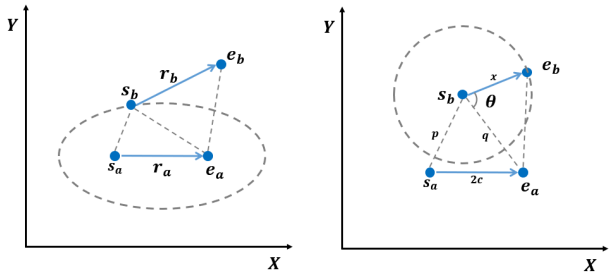
Given two requests r_a and r_b, they are *shareable* if there exist at least one feasible route for them to share.

Definition 5 (Shareability Graph)**.** Given a set of m requests R, let $SG = \langle R, E\rangle$ denote a *shareability graph* with a set of nodes R and edges E, where each node in R indicates a request. Edge $(r_a, r_b) \in E$ denotes r_a and r_b are shareable.

For example, in the *shareability graph* shown in Figure 1(b), the edge $e = (r_1, r_2)$ between r_1 and r_2 reflects that there exist one feasible route to serve them in a trip (e.g., a schedule of $\langle s_1, s_2, e_2, e_1\rangle$). Thus, we can easily find out the candidate shareable requests for each request by retrieving the neighbors in the shareability graph. Moreover, we have the following two observations:

Observation 1. *The degree of each request in the shareability graph reveals its shareability and importance in the corresponding batch.* We can intuitively evaluate the sharing opportunities of requests by comparing their degrees in the shareability graph. We call the degree of a request as its *shareability*. For example, a request with a smaller shareability is often more urgent to select an appropriate shareable request.

Observation 2. *The potential shareable requests tend to be denser in the shareability graph.* In graph databases, *Clique Detection* is a popular community detection problem, which shows the tight relationship between nodes in a graph [17], [37], [38]. In the shareability graph, by Theorem III.1, if there exists a valid sharing schedule consists of k requests, there must exist a *k-clique* among the corresponding nodes. With this observation, we can prune some infeasible combinations in the grouping phase of each batch.



(a) Ellipse Pruning Example (b) Angle Pruning Example

Fig. 3: An Illustration Example for *Shareability Graph Builder*

Theorem III.1. *Let S be a valid sharing schedule for k requests, then the corresponding nodes of k requests in the shareability graph form a k-clique.*

Proof. We prove it by contradiction. For a set, R, of k requests, there exists a valid sharing schedule S consisting of their sources and destinations. Assume the corresponding nodes of k requests in shareability graph do not form a k-clique, thus at least two requests, $r_a, r_b \in R$, are not connected.

Let S' be a subsequence of S by removing all other requests' sources and destinations except for r_a and r_b. Removing the waypoints from S will reduce the detours and therefore keep the validation of S. Thus, S' must still be a valid schedule. With the shareability graph's definition, an edge must exist connecting r_a and r_b in the corresponding shareability graph, which contradicts our assumption. In conclusion, the nodes of k requests form a k-clique in the shareability graph. □

B. Ellipse and Angle Pruning Strategies

To build the shareability graph for each batch $\mathcal{P}_k$, the basic idea is to enumerate all pairs of requests (r_a, r_b) in $\mathcal{P}_k$ and check there exists a valid sharing schedule with a linear insertion method. The number of times to call the shortest path query in such a process determines the efficiency of the construction algorithm. To avoid enumerating all pairs of requests, we proposed two pruning strategies, ellipse pruning and angle pruning illustrated in Figure 3, for more efficient shareability graph construction. Note that, when we prune the candidate requests for a given request r_a, to avoid duplicated consideration of schedules, we only consider the schedules with s_a (the source of r_a) as the first waypoint.

Ellipse pruning. This pruning strategy is based on the source locations of the candidate requests, as we observe that the shareable requests share similar sources. If the sources of requests are far from each other, it's hard for drivers to deliver these requests on time in one trip due to the constraint of maximum waiting time. For each request r_a, we prune the requests whose sources locate out the ellipse of r_a defined in Theorem III.2 to avoid unnecessary shortest path queries.

Theorem III.2. *For a request r_a, the source s_b of its shareable request r_b must fall into the ellipse with s_a and e_a as the focus and $d_a/2$ as semi-major axis, where $cost(s_a, s_b) + cost(s_b, e_a) \leq d_a$.*

Proof. For any candidate shareable request r_b, only two feasible paths are starting from s_a: (a) $\langle s_a, s_b, e_a, e_b\rangle$; (b)

$\langle s_a, s_b, e_b, e_a\rangle$. The earliest arriving time of e_a is contributed by (a), which denotes as $arrive(e_a) = cost(s_a, s_b)+cost(s_b, e_a)$. Recall that r_a must be delivered to destination before deadline d_a, thus, $arrive(e_a) = cost(s_a, s_b) + cost(s_b, e_a) \leq d_a$. This completes the proof. □

Angle pruning. Our second pruning strategy is based on the travel directions of the requests. We notice that the requests with similar travel directions have a higher probability of sharing the trips. For instance, since the driver needs to turn around and spend much time driving in the opposite direction to deliver the other passenger after finishing one of them, a northward request is hard to share with a southward request. We defined a threshold δ to prune request pairs that have similar sources but divergent directions with Theorem III.3. Let $\overrightarrow{ab}$ be the vector pointing from a to b.

Theorem III.3. *For a request r_a and its candidate sharing request r_b, the expected probability of sharing a trip will increase when the angle θ between $\overrightarrow{s_be_a}$ and $\overrightarrow{s_be_b}$ decreases.*

Proof. Firstly, we denoted the maximum detour time for request r_a as $(\gamma - 1) \times cost(s_a, e_a)$. For the two possible schedules of r_a and its candidate request r_b ($\langle s_a, s_b, e_a, e_b\rangle$ or $\langle s_a, s_b, e_b, e_a\rangle$), one of the following two conditions needs to be satisfied due to the deadline constraint: (a) $p + x + cost(e_a, e_b) \leq 2\gamma c$; (b) $p + q + cost(e_a, e_b) \leq \gamma x$, where p, q, x and $2c$ indicate $cost(s_a, s_b)$, $cost(s_b, e_a)$, $cost(s_b, e_b)$ and $cost(s_a, e_a)$, respectively.

When request r_a, source s_b and $cost(s_b, e_b)$ are fixed, the total travel cost of the two possible schedules only depends on $cost(e_a, e_b)$. Then, $cost(e_a, e_b)$ can be represented through x, n and $\cos(\theta)$. In (a), we have $x \leq \frac{(2\gamma c-p)^2-q^2}{4\gamma c-2p-2q\cos(\theta)} \leq 1/(\frac{cos^2(\theta/2)}{(\gamma+1)c} + \frac{sin^2(\theta/2)}{(\gamma-1)c})$ (noted as $g(c)$), because $p+q \leq 2(\gamma+1)c$ and $p - q \leq 2(\gamma - 1)c$ hold in the ellipse in Theorem III.2. In (b), we have $\gamma x \geq p + q + \sqrt{(x - q * cos(\theta))^2}$, which can be rewritten as $x \geq \frac{2c(1-cos(\theta))}{\gamma-1}$ (marked as $h(c)$) by $p+q \geq 2c$. In both cases, the feasible range of x gradually decreases as the angle reduces. In other words, for a given candidate request r_b, when the angle θ between $\overrightarrow{s_be_a}$ and $\overrightarrow{s_be_b}$ decreases, the travel cost $x = cost(s_b, e_b)$ is more likely to satisfy the deadline constraint. □

By investigating the real datasets of Chengdu and New York City in Section V, we found that the distances of the requests almost follow the log-normal distribution whose probability density function is $f(x; \mu; \sigma) = \frac{1}{\sqrt{2\pi}x\sigma} e^{-\frac{(\ln x-\mu)^2}{2\sigma^2}}$, where μ and σ are two parameters of log-normal distribution. Therefore, the expected probability of r_a to share with a candidate request r_b with an angle θ greater than a given threshold δ can be evaluated as follows:

$$\mathbb{E}(\theta \geq \delta) = \int_0^{+\infty} f(x) \left(\int_0^{g(\frac{\pi}{2})} f(y)dy + \int_{h(\frac{\pi}{2})}^{+\infty} f(y)dy \right) dx$$

For instance, we fit the probability density function of the requests' distances of Chengdu and NYC datasets with a log-normal distribution. The probability expectations $\mathbb{E}(\theta \geq \frac{\pi}{2})$ are 40.98% and 41.38% respectively when $\gamma = 1.5$. Note that the requests we have ignored are either too short or hard to serve (i.e., long detour).

Algorithm 1: *Shareability Graph* Builder

Input: A set R of n requests with an angle threshold δ
Output: The corresponding shareability graph SG

1 $V = R$ and $E = \emptyset$
2 **for** $r_a \in R$ **do**
3 $C \leftarrow$ candidate requests filtered by Theorem III.2
4 **for** $r_b \in C$ **do**
5 **if** $\arccos\left(\frac{\overrightarrow{s_be_a}\cdot\overrightarrow{s_be_b}}{|\overrightarrow{s_be_a}||\overrightarrow{s_be_b}|}\right) \in \left[-\frac{\delta}{2}, \frac{\delta}{2}\right]$ **then**
6 **if** *r_a and r_b are shareable* **then**
7 $E = E \cup (r_a, r_b)$
8 **return** $SG = \langle V, E\rangle$

C. Shareability Graph Builder

The details of the shareability graph builder algorithm are illustrated in Algorithm 1. Firstly, we initialize all requests as nodes in the shareability graph (line 1) and connect the shareable nodes gradually (lines 2-7). We try to find the shareable relationships for each node successively (lines 2-7). With the benefit of Theorem III.2, we can obtain a candidate request set C which shares a similar source location with r_a by the constant time calculation instead of the shortest path query. After that, we further identified the requests $r_b \in C$ by angle pruning rule (lines 4-7). We construct the vector $\overrightarrow{s, e}$ by the source and destination of the corresponding request to represent the distance and direction. If the angle θ of $\overrightarrow{s_be_a}$ and $\overrightarrow{s_be_b}$ is out of the given angle threshold δ, r_b will be pruned (line 5). We'll add an edge (r_a, r_b), if r_a and r_b are shareable (lines 6-7). Finally, we get a shareability graph SG composed of V and E generated in the above steps (line 8).

Complexity Analysis. Assume that the shortest path query takes $O(q)$ time. For a given request r_a, we can complete the filtering step within constant time by searching *Minimum Bounding Rectangle* of the ellipse in grid index. However, in the worst case, the size of $|C| = n - 1$ in line 3. Besides, the angle calculation only takes constant time, so that we only need to perform two times insertion when testing whether r_a and r_b are shareable in $O(k)$. Thus, the final complexity of the algorithm is $O(q \cdot n^2)$.

Discussion. The ellipse and angle pruning strategies proposed in this section are derived based on Euclidean space. In a realistic road network, facilities such as expressways may exist, making some feasible solutions excluded. However, we conclude in our experiments that only very few cases will be discarded, so it is acceptable to prune the shareability network heuristically with such an approach.

IV. Structure-Aware Dispatching

With the shareability graph, we can intuitively analyze the shareability of each request and propose the structure-aware ridesharing dispatching (SARD) algorithm. Specifically, we first discuss two main schedule maintenance methods. Then we introduce a bottom-up enumeration strategy for the different combinations of requests.

A. Schedule Maintenance

There are two state-of-art strategies for schedule maintenance in previous work: kinetic tree insertion [6] and linear

insertion [25], [32]. The kinetic tree maintains all feasible schedules and checks all available waypoints ordering exhaustively to insert a new request. Even kinetic tree can get the *exact* optimal schedule for the vehicle, but it needs to maintain up to $\frac{(2m)!}{2^m}$ schedules in the worst case (m is the number of requests assigned in the vehicle). In contrast, the schedule obtained by the linear insertion method is optimal only for the current schedule (i.e., local optimal). Linear insertion is optimal when the number of requests is 2. In the experimental study in Section V, we find that in NYC and Chengdu datasets, if we use linear insertion to handle requests according to the release time of the request, the probability of achieving a global optimal schedule is up to 89% and 85%, when we insert the third and fourth request, respectively. To improve such a probability with the linear insertion method, we reorder the insertion sequence of the requests based on Observation 1 in Section III-A. Specifically, we first select two requests with the lowest shareability (i.e., the degree of the node) in the shareability graph and generate an optimal sub-schedule. Then, we insert the remaining requests into the sub-schedule one by one in the ascending order of their shareability. In this way, we improve the probability of achieving optimal schedule using linear insertion method up to 91% and 90%.

B. Request Grouping

In batch mode, we need to enumerate all feasible request combinations before the assignment phase. If we simply construct groups by enumerating all possible combinations, we need to check $\sum_{i=1}^{c}\binom{n}{i}$ groups, where c is the capacity of vehicles. After that, we need to verify the existence of a feasible route to serve these requests by linear insertion [25] in $O(k^2)$ for each group, where k is the size of the group. However, many groups are invalid for vehicles in practice. To avoid unnecessary enumeration, Zeng *et al.* proposed an index called *additive tree* [29], which enumerates valid groups level by level through a tree-based structure. In the additive tree, each node represents a valid group of requests, and the group for every node is an extension group from the group of its parent node through adding one more request.

Although the additive tree can help prune some invalid groups in the enumeration, it still needs to maintain all valid schedules for each tree node. However, we will only pick the best one of these schedules for each group at the end. With the schedule maintenance method proposed in Section IV-A, we can heuristically reorder the insertion sequence to get an optimal schedule with higher probability. In building the modified additive tree, we only keep one schedule for each node by inserting one new request to its parent's schedule.

The detailed construction steps are shown in Algorithm 2. Firstly, we initialize c empty sets $\{RG_1, RG_2, \cdots, RG_c\}$ to store c levels of nodes for the modified additive tree (line 1). Then, we construct the groups formed by individual request $r_a \in R$ whose schedule consists of its source and destination for level 1 of the modified additive tree (lines 2-3). For the construction of the feasible groups of the remaining levels RG_l, we generate each node in level l by traversing and merging pairs of parent nodes' groups RG_{l-1} (lines 4-11).

Algorithm 2: Request Grouping Algorithm

Input: A *shareability graph* SG of a batch instance and a set R of n requests, the capacity of vehicles c
Output: Request groups set $\mathcal{RG}$.

```
1  initialize {RG_1, RG_2, ..., RG_c} as empty sets
2  forall r_a ∈ R do
3      RG_1.insert({r_a}, ⟨s_a, e_a⟩)
4  for l ∈ [2..c] do
5      foreach pair (G_x, G_y) in RG_{l-1} do
6          G ← G_x ∪ G_y
7          if |G| = l and G satisfied Lemma IV.1 then
8              r_b ← find maximum degree node in G
9              S ← insert r_b into the schedule S' of G \ {r_b}
                 maintained in level RG_{l-1}
10             if S' is valid then
11                 RG_l.insert(G, S)
12 return RG = {RG_1, RG_2, ..., RG_c}
```

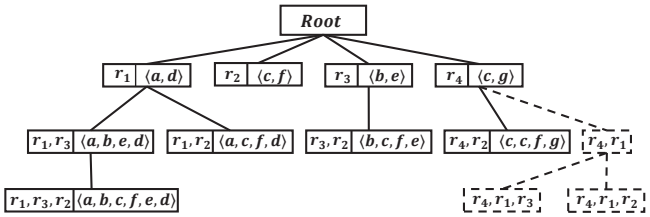


Fig. 4: Grouping Tree in Example 2.

During constructing the nodes in level l, we only consider the groups G with l requests and G must satisfy Lemma IV.1 (line 7). For each feasible group G, we search for the request $r_b \in G$ with the maximum shareability in SG (line 8). Based on the schedule S' of its parent node for group $G \setminus \{r_b\}$, we generate the new schedule S of G through inserting r_b to S' with the linear insertion method (line 9). If the generated schedule is valid for G, we store the new group with its maintained schedule (G, S) into RG_l (lines 10-11).

Lemma IV.1. *For any valid group G_x: (a) $\forall r_a \in G_x$, the group $G_x \setminus \{r\}$ must be also valid; and (b) the nodes of $r_a \in G_x$ forms a clique in the shareability graph.*

Proof. The first condition has been proved by Lemma 2 in [29] and condition (b) is derived from Theorem III.1. □

Example 2. *Consider the example in Figure 1. We first initialize the groups composed of a single request with corresponding schedule. Then, we enumerate all pairs of 1-size groups and merge them to generate a new schedule for child nodes. Because $deg(r_2) > deg(r_3)$ in shareability graph SG in Figure 1(b), we maintain the schedule for group $\{r_3, r_2\}$ by inserting r_2 into the schedule of group $\{r_3\}$. The schedule generated by linear insertion method is optimal when the size of the group is 2. At the same time, we take the generated group $\{r_3, r_2\}$ as a child of group $\{r_3\}$. Since we cannot find a valid schedule for r_4, r_1, all groups containing $\{r_1, r_4\}$ are pruned in subsequent steps. Next, we insert r_2 into the schedule of group (r_1, r_3) and get an approximate schedule $\langle a, b, c, f, e, d\rangle$ because $deg(r_2) > deg(r_1) = deg(r_3)$. Since we cannot find any valid group G which $|G| \geq 4$, we terminate the group building process and got the final result in Figure 4.*

Complexity Analysis. In the worst case, all the requests in a batch can be arbitrarily combined, so that there are at most $\sum_{i=1}^{c} \binom{n}{i}$ nodes in total. Since the capacity constraint $c \ll n$ in practice, the number of combination groups can be noted as $O(n^c)$. We only need $O(c)$ time for each new group to perform linear insert operation for a new schedule. We can finish the request grouping in $O(c \cdot n^c)$.

C. SARD Algorithm

With the grouping algorithm in Section IV-B, we proposed a two-phase matching algorithm, *SARD*, to solve the matching between vehicles and requests in this section. The intuition of SARD is to greedily maintain the shareability or connectivity of the nodes in the shareability graph such that the requests have higher rates to share with other requests in the final assignment and schedules. In SARD, requests at the beginning proposed to *worse* vehicles (with larger increases of travel costs), which can give more initiative to vehicles on selecting groups of requests. Then, for each vehicle v_j, it greedily accepts a group of proposed requests with the smallest *shareability loss*, and the rejected requests will propose to other better vehicles in the following round proposal. The proposal and acceptance phases are iteratively conducted until no request will propose. We first define shareability loss, then introduce two theorems to support our design of SARD.

To define shareability loss, we introduce a substitution operation to replace a k-clique G_i in shareability graph SG with a supernode $\hat{v}_i$. After we substitute a supernode $\hat{v}_i$ for a k-clique G_i, there is an edge connecting another node $v_j \in SG \setminus G_i$ with $\hat{v}_i$ if and only if v_j connects to every node of G_i in the original shareability graph. Then, we define shareability loss as:

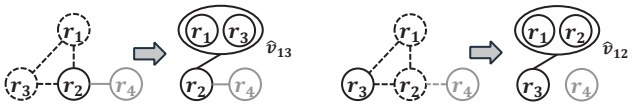
Definition 6 (Shareability Loss)**.** Given a shareability graph $SG = \langle V, E \rangle$, the shareability loss $SLoss(G_i)$ of substituting a super-node $\hat{v}_i$ for a k-clique group $G_i \subseteq V$ is evaluated with the following structure-aware loss function:

$$SLoss(G_i) = \max_{r \in G} \{ | \bigcap_{v \in G - \{r\}} N(v)| + |N(r)| - | \bigcap_{v \in G} N(v)| - 1\}, \quad (5)$$

where $N(v)$ is the set of neighbor nodes of v in the shareability graph SG. $SLoss(G) = deg(r)$ if $|G| = |\{r\}| = 1$.

Example 3. *With the shareability graph in Figure 1(b), we illustrate the idea of shareability loss in Figure 5, where we assume r_4 is not available. In Figure 5(a), we try to merge r_1 and r_3 into a supernode as follows. We first remove the edges incident to r_1 and r_3. Since r_1, r_2 and r_3 form a 3-clique in the original graph, there should be a shareable relation between r_2 and the supernode $\hat{v}_{13} = \{r_1, r_3\}$ by Theorem III.1. Thus, we add a new edge between r_2 and $\hat{v}_{13}$. Overall, if we substitute $\hat{v}_{13}$ for $\{r_1, r_3\}$, the shareability loss is $SLoss(\{r_1, r_3\}) = 3 - 1 = 2$. In Figure 5(b), we try to merge r_1 and r_2 into a supernode $\hat{v}_{12} = \{r_1, r_2\}$. Similarly, four edges incident to r_1 and r_2 are removed, and a new edge between $\hat{v}_{12}$ and r_3 is built. Then, the shareability loss is $SLoss(\{r_1, r_2\}) = 4 - 1 = 3$ here. Therefore, substituting $\{r_1, r_3\}$ is more structure-friendly than substituting $\{r_1, r_2\}$.*

Shareability loss can guide a vehicle to select a set of requests to serve from the potential requests (i.e., proposed to the vehicle in the proposal phase). Specifically, a vehicle v_j should select a set of requests whose shareability loss is the minimum among all groups of its potential requests. In Theorem IV.1, we prove that through serving a group of requests with the minimum shareability loss, the remaining requests can have a higher upper bound of sharing rate.



(a) $SLoss(\{1, 3\})$ (b) $SLoss(\{1, 2\})$

Fig. 5: An Illustration Example of Shareability Loss

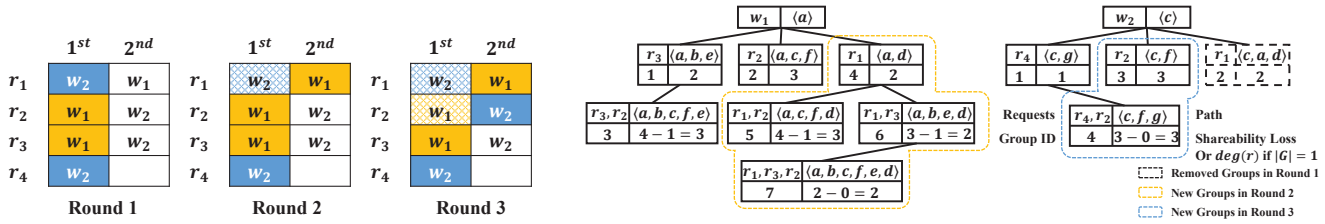
Theorem IV.1. *Substituting a supernode for group G with smaller shareability loss will increase the upper bound of sharing probabilities more for the remaining nodes in the shareability graph.*

Proof. According to Theorem III.1, all valid groups we picked in the assignment phase always constitute a k-clique in the shareability graph. Therefore, we model the problem of maximizing sharing probability as a problem of clique partition [39], which tries to find the minimum number of cliques to cover the graph such that each node appears in precisely one clique. In the worst case, we will partition each node in the shareability graph into a 1-clique, and the sharing probability of the nodes is 0 (i.e., no requests can share). Intuitively, the less clique we used to cover the shareability graph, the higher the sharing probability is. Although the Theorem III.1 provides only a necessary but not sufficient condition for a clique to be a shareable combination, the clique in the shareability network still largely accounts for the potentially high probability of these requests to be shareable. Hence, in the following proof, we illustrate the help of the shareability loss on the sharing probability by minimizing the number of cliques. Moreover, since the vehicle's capacity limits the size of each group in our problem to k, we consider the problem of partitioning the shareability graph into a minimum number of cliques of size no more than k. In [40], J. Bhasker *et al.* present an optimal upper bound (shown in equation 6) for the clique partition problem evaluated with the number of nodes n and the number of edges e for the graph.

$$\theta_{upper} = \lfloor \frac{1 + \sqrt{4n^2 - 4n - 8e + 1}}{2} \rfloor \quad (6)$$

In addition, we observed that the degrees of most riders in the shareability graph are relatively small, which is consistent with a power-law distribution. For analytical tractability, we assume that the degree of the node in the shareability graph as a random variable δ following the power-law distribution with exponent η, which complementary cumulative distribution is shown as $\Pr(\delta \geq x) = ax^{-\eta}$. For a shareability graph SG with n nodes and the degree of its every node follows the power-low distribution with exponent η, Janson *et al.* [41] reveals that the size of the largest clique $\omega(SG(n, \eta))$ in graph SG is a constant with $\eta > 2$. However, in the heavy-tail distribution, $\omega(SG(n, \eta))$ grows with $n^{1-\eta/2}$ (the formal description is shown in equation 7).

$$\omega(SG(n, \eta)) = \begin{cases} (c + o_p(1))n^{1-\eta/2}(\log n)^{-\eta/2}, & \text{if } 0 < \eta < 2 \\ O_p(1), & \text{if } \eta = 2 \\ 2 \text{ or } 3 \text{ w.h.p}, & \text{if } \eta > 2 \end{cases} \quad (7)$$



(a) Transformation of the Candidate Queue

(b) Grouping Trees for Vehicles

Fig. 6: An Example for the *SARD* Algorithm

For any shareability graph SG, we deal with the optimal partition $\{C_1 \ldots, C_\theta\}$ of general clique partition problem on SG as follows: for each clique C_i, we divided C_i into sub-cliques with size no larger than k. After that, we obtained an upper bound θ'_{upper} after scaling of our problem with n and e by formula 6 and 7.

$$\theta'_{upper} = \lfloor \frac{1 + \sqrt{4n^2 - 4n - 8e + 1}}{2} \rfloor \cdot \lceil \frac{\omega(SG(n, \eta))}{k} \rceil \tag{8}$$

The higher the number of edges e in the shareability graph SG is, the lower the upper bound θ'_{upper} of the number of clique partitions is. In conclusion, substituting a supernode for the group of nodes with the lowest edge loss will keep the most edges left in the shareability graph SG, improving the remaining nodes' sharing probabilities in SG. This completes the proof. □

To merge nodes whose degrees are 2 in the shareability graph at the beginning, we have the following theorem:

Theorem IV.2. *Given a shareability graph $SG = (V, E)$, merging the node v, whose degree is 1, with its neighbor into a 2-clique will not reduce the pairing rate in SG.*

Proof. Suppose that the optimal group partitions with size no larger than k on SG are $\{G_1, \ldots, G_n\}$, and v_x and v_y are included in G_a and G_b, respectively. The size of the clique $|G_a| \leq 2$ because $deg(v_x) = 1$. If v_x and v_y are included in different partitions ($a \neq b$), the size of the clique $|G_a| = 1$ because v_x is not connected to any node except v_y. Thus, removing v_y from G_b and merging it into G_a will not increase the number of isolated groups with a size no more than 1. Otherwise, ($a = b$), v_x and v_y are included in the same partition and forms a 2-clique, which is consistent with the theorem. Thus, the operation in the theorem will not violate the pairing rate of nodes on SG. This completes the proof. □

With Theorems IV.1 and IV.2, we design our SARD, whose pseudo code is shown in Algorithm 3. Firstly, we maintain a current working set R_p (line 1), which contains all available requests until the current timestamps. When each batch arrives, we insert the requests in this batch into R_p (line 3) and process them together with the available (unmatched and unexpired) requests in the previous round (lines 2-18). Before starting the two-phase processing, we retrieve all candidate vehicles for each request r_a and store them into a priority queue $\mathcal{Q}_{r_a}$ in descending order of the increased utility cost for serving r_a (lines 5-7). In the proposal phase (lines 9-11), the discarded requests in the previous round will propose to its current worst vehicle (i.e., the vehicle with the highest increase in travel cost on serving the request) (line 11). After that, we enumerate

Algorithm 3: *SARD*

Input: A set R of n requests with a batching time period T and a set W of m vehicles
Output: A vehicle set W with updated schedules.

1 $R_p \leftarrow \emptyset$ ▷ *for unmatched and unexpired requests*
2 **foreach** *batch $\mathcal{P}$ within time period T* **do**
3 insert $r_a \in \mathcal{P}$ into working pool R_p
4 $SG \leftarrow$ build *shareability graph* for W with Algorithm 1
5 **foreach** $r_a \in R_p$ **do**
6 $\mathcal{Q}_{r_a} \leftarrow$ initialize a priority queue by Δ utility
7 insert candidate vehicles into $\mathcal{Q}_{r_a}$
8 **while** $\exists r_a \in R_p, |\mathcal{Q}_{r_a}| \neq 0$ **do**
9 **foreach** $r_a \in R_p$ ***and*** $|\mathcal{Q}_{r_a}| \neq 0$ **do** ▷ *Proposal Phase*
10 $w_x \leftarrow \mathcal{Q}_{r_a}$.pop() the worst vehicle
11 R_{w_x}.insert(r_a)
12 **foreach** $w_x \in W$ **do** ▷ *Acceptance Phase*
13 $G_{w_x} \leftarrow$ grouping R_{w_x} by Algorithm 2
14 $G^*_{w_x} \leftarrow$ select a group with minimum shareability loss from G_{w_x}
15 push $w_x.ac \setminus G^*_{w_x}$ back to R_p for next proposal
16 **foreach** $r_b \in G^*_{w_x}$ **do**
17 remove r_b from R_{w_x} and add r_b to $w_x.ac$
18 remove expired requests from R_p
19 **return** W

the feasible groups of requests received by each vehicle w_x by Algorithm 1. We evaluate the validation of group nodes based on w_x's current schedule (line 13). Specifically, we replace the single request schedule in line 3 of Algorithm 1 with the schedule generated by inserting the request into the worker's current schedule, which helps to filter out the groups that cannot be served by vehicle w_x. With Theorem IV.1, we prioritize the groups with minor shareability loss. We select a set of groups with the minimum shareability loss for vehicle w_x(line 14). Then, we put the discarded requests back to the working pool R_p (line 15), where $w_x.ac$ indicates the currently accepted requests of vehicle w_x. At the end of each acceptance phase, we assign the selected group to the vehicle and put the rejected and evicted requests back into the working set R_p for their subsequent proposal (lines 16-17). We repeat the proposal and acceptance phase until no request will propose. Finally, we remove the expired requests from R_p that cannot be completed due to exceeding the maximum waiting time at the end of each batch.

Example 4. *Let's consider the batch of requests in Example 1. Suppose there are two idle vehicles, w_1 and w_2, located at a and c. We first construct a candidate vehicle queue for $r_1 \sim r_4$. For instance, the travel cost of w_1 and w_2 for serving r_1 is $cost(r_1)$ and $cost(r_1) + cost(c, a)$, respectively. The priority*

TABLE III: Experimental Settings.

Parameters	Values
the number, n, of requests	10K, 50K, 100K, 150K, 200K, **250K**
the number, m, of vehicles	1K, 2K, 3K, 4K, **5K**
the capacity of vehicles c	2, 3, **4**, 5, 6
the deadline parameter γ	1.2, 1.3, **1.5**, 1.8, 2.0
the penalty coefficient p_r (β)	2, 5, **10**, 20, 30
the batching time Δ (s)	1, 3, **5**, 7, 9

sequence in r_1's candidate queue is $\langle w_2, w_1 \rangle$. We can calculate the candidate queue of the remaining requests, as shown in Figure 6(a). In the first round, r_2 and r_3 are put into the candidate pool of w_1, while the remaining requests are put into the candidate pool of w_2. Then, w_1 and w_2 enumerate all the groups by Algorithm 2 with the requests in their candidate pool (as shown in Figure 6(b), except for the colored dashed area). In the grouping of w_1, we take $\{r_3, r_2\}$ as the temporal assignment because it's the only group whose size is not less than 2. Since there are no groups whose size is not less than 2 in the first round of w_2, we prefer the group $\{r_4\}$, which has a lower degree by default as it leads to less shareability loss. Since r_1 is not accepted by vehicle w_2, it will propose to w_1 according to its candidate list in the second round. At this time, the w_1's grouping tree will update to the groups shown in the Figure 6(b). Although the loss of G_6 and G_7 are same, w_1 will update its current best group to $\{r_1, r_3\}$ rather than $\{r_1, r_3, r_2\}$, because the planned route of G_6 holds higher sharing ratio $\frac{cost(P)}{\sum_{r \in SG} cost(r)}$. Therefore, r_2 will be discarded and propose in the third round, and w_2 will take $\{r_4, r_2\}$ by Theorem IV.2. Finally, w_1 and w_2 was assigned with $\{r_1, r_3\}$ and $\{r_2, r_4\}$, respectively.

Complexity Analysis. We need to perform an insertion operation in $O(c)$ time to obtain the increased travel cost. Thus, the time cost of constructing candidate queues for a batch of size n is $O(c \cdot n^2)$. In the proposal phase, every request will propose to every car at most one time in the worst case. Thus, we need to build a grouping tree of the whole batch for each car in $O(m \cdot n^c)$. The time complexity of *SARD* is $O(c \cdot n^2 + m \cdot n^c)$ in total.

V. Experimental Study

A. Data Set

The road networks of *CHD* and *NYC* are retrieved as directed weighted graphs shown in Table IV. The nodes in the graph are intersections, and the edges' weight represents the required travel time on average. The shortest path query on the road network adopts the hub labeling algorithm [42] with LRU cache [43] for all algorithms.

We conduct experiments on two real datasets [25]. The first dataset is the requests collected by Didi in Chengdu, China (noted as *CHD*). The second one is the yellow and green taxi in New York, USA (noted as *NYC*). These datasets contain the following information about the request: release time, longitude and latitude of the source location and destination, and the number of passengers. For requests with multiple riders, they will be examined during the group enumeration in Algorithm 2. We extracted the requests of the day with the largest number of requests as our request data (November 18, 2016, and April 09, 2016). We set the deadline of request r_i as $d_i = t_i + \gamma \cdot cost(r_i)$, which is a commonly used configuration

TABLE IV: Details of Road Networks.

Name	#Nodes	#Edges	#Requests
CHD	214,440	466,330	259,347
NYC	112,572	300,435	424,635

in many existing works [43], [26], [44]. We set the maximum waiting time threshold for requests to be 5 minutes, which means $w_i = \min(5min, d_i - cost(r_i) - t_i)$. The distribution of the sources and destinations of the datasets are shown in Figure 9. We map the sources and destinations of the requests to the corresponding nodes of the road network in advance. The initial positions of the vehicles are chosen randomly from the road network. The number of vehicles varies according to the experimental settings. In the following experiments, we use CHD's data for a whole day and NYC for half a day (with about 250K requests for each dataset). We set the parameter of the utility function $\alpha = 1$ and the default value of β to 10 as in [25]. Table III shows the parameter settings, and the default values are in bold.

B. Approaches and Measurements

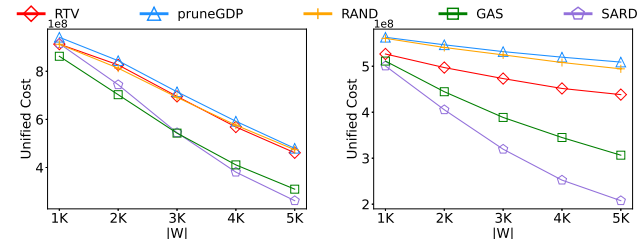
We compare our SARD with the following algorithms:

- ***pruneGDP [25].*** It inserts the request into the vehicle's current schedule sequentially and selects the vehicle with the least increased distance for service.
- ***RTV [28].*** It is in batch mode and obtains an allocation scheme with the least increased distance and most serving vehicles by linear programming.
- ***GAS [29].*** It is in batch mode and enumerates all combinations for each vehicle's candidate requests in random order. It selects the most profitable group for each vehicle. The length of total requests in each group is used as the profit.
- ***RAND.*** It randomly arranges requests to a feasible vehicle sequentially, and the vehicle's trip schedule is maintained by the linear insertion method [25].

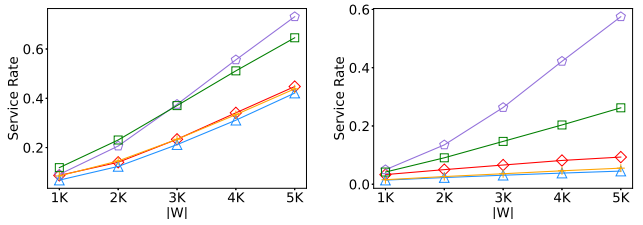
We report the unified cost, service rate, and overall running time of all algorithms. Specifically, the unified cost adopts the evaluation of total revenue in [25], and the varying penalty coefficient p_r is equivalent to the balance between income per unit time and fare per unit distance. Additionally, the response time of a single request can be calculated from the running time divided by the number of batches, and in the experiments, the responses to requests are two milliseconds on average for SARD. We conduct experiments on a single server with Intel Xeon 4210R@2.40GHz CPU and 128 GB memory. All the algorithms are implemented in C++ and optimized by -O3 parameter. Besides, we use *glpk* [45] to solve the linear programming in the RTV algorithm, and all algorithms are implemented in a single thread.

C. Experimental Results

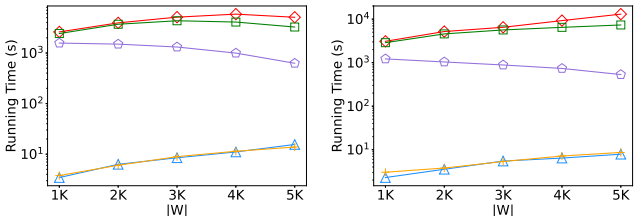
Effect of the number of vehicles. Figure 7 shows the results of varying the number of vehicles from 1K to 5K. As the number of vehicles increases, so does the service quality of the evaluated methods. For the unified cost, SARD and GAS are leading the other methods, and the margin between these two algorithms and the others are widening gradually. In the *CHD* dataset, the results of the two algorithms are very close, but SARD achieves an improvement of $2.09\% \sim 59.22\%$



(a) Unified Cost (*CHD*) (b) Unified Cost (*NYC*)

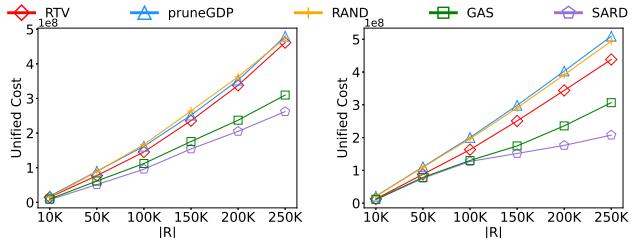


(c) Service Rate (*CHD*) (d) Service Rate (*NYC*)

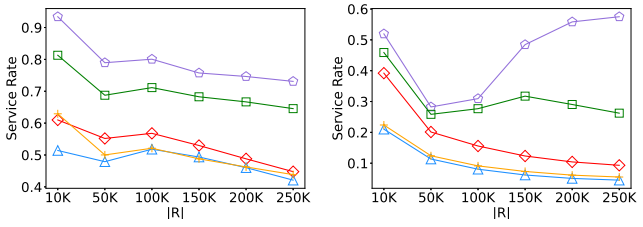


(e) Running Time (*CHD*) (f) Running Time (*NYC*)

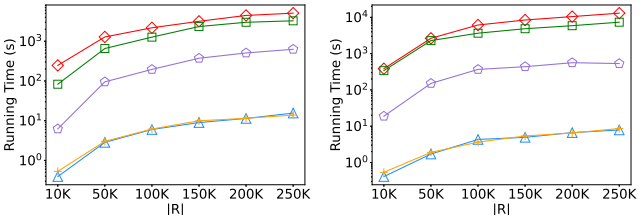
Fig. 7: Performance of Varying $|W|$.



(a) Unified Cost (*CHD*) (b) Unified Cost (*NYC*)

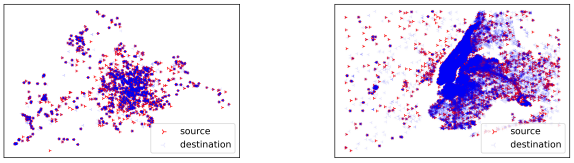


(c) Service Rate (*CHD*) (d) Service Rate (*NYC*)



(e) Running Time (*CHD*) (f) Running Time (*NYC*)

Fig. 8: Performance of Varying $|R|$.



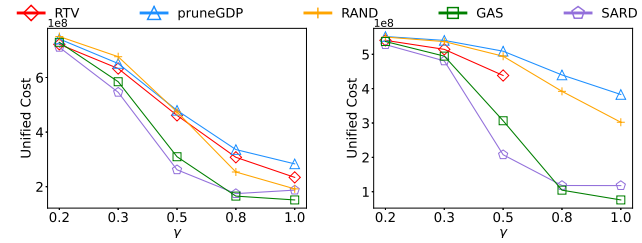
(a) Requests distribution (*CHD*) (b) Requests distribution (*NYC*)

Fig. 9: The distribution of datasets.

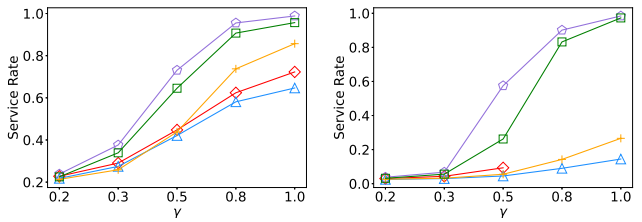
compared with other methods in the *NYC* dataset. As for the service rate, since the penalty part of the unified cost caused by the rejected requests decreases when the algorithm achieves a higher overall service rate, the gap between the algorithms is consistent with the trend of the unified cost. Compared with other existing methods, our method obtains an improvement on service rate up to 30.99% and 52.99% on two datasets, respectively. pruneGDP and RAND are the winners in terms of the running time due to the efficiency of linear insertion. RTV, GAS, and SARD are slower than pruneGDP. Specifically, SARD achieves a speedup ratio up to $7.12\times$ and $23.5\times$ than RTV and GAS on two datasets, respectively. In addition, the running time of SARD even decreases with the increase of vehicle number, which is mainly due to the decrease in the number of propose-acceptance rounds. When m increases (more vehicles available), fewer riders will conflict, and thus fewer rounds of the proposal are needed. Moreover, the average number of proposed riders will decrease for each vehicle, then the time of acceptance stage also decreases.

Effect of the number of requests. Figure 8 presents the results of varying the number of requests from 10K to 250K. Because the number of accepted and rejected requests increased significantly with the increase of requests number, the unified costs of all experiment algorithms are growing. Meanwhile, SARD and GAS achieve smaller unified costs than other methods when the number of requests becomes larger. For service rate, SARD improves the service rate by $41.97\% \sim 52.99\%$ at most compared with pruneGDP. Besides, compared with the state-of-art batch-based method GAS, SARD achieves up to 12.09% and 31.27% higher service rates on the two datasets, respectively. For the running time, the insertion-based methods are still faster. But among the batch-based methods (i.e., SARD, RTV and GAS), SARD is $4.21\times \sim 38.6\times$ faster than GAS and RTV on two datasets.

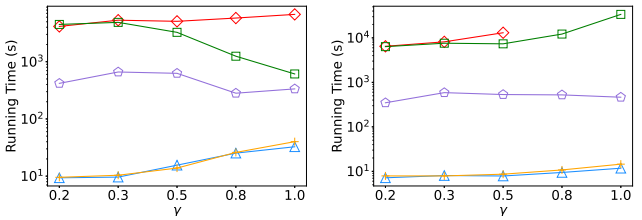
Effect of deadline. Figure 10 presents the results of the varying deadline of requests by changing the deadline parameter γ from 1.2 to 2.0. For service rate, the results of SARD are similar to the existing algorithms when we set the deadline of requests strictly, i.e., $\gamma = 1.2$. The reason is that the number of candidate vehicles for each request reduces significantly with a minor deadline, making it challenging to achieve noticeable performance improvements by applying grouping strategies in batch mode. With the increase of deadline, the superiority of SARD and GAS gradually realizes. The service rate of SARD is more than 90% when the deadline is $1.8\times$, which is up to 37.42% higher than other existing algorithms on *CHD* dataset. The superiority of SARD is more explicit in the *NYC* dataset, where the service rate of SARD is up to 83.98% higher than that of pruneGDP. GAS performs inefficiently on *NYC* dataset with $\gamma = 2.0$, which is primarily due to the increase of request combinations with the relaxation of the deadline. Besides, GAS enumerates all the combinations of requests and schedules almost for each vehicle. However, in SARD, the



(a) Unified Cost (*CHD*) (b) Unified Cost (*NYC*)

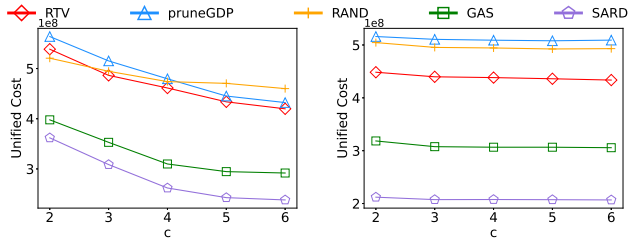


(c) Service Rate (*CHD*) (d) Service Rate (*NYC*)

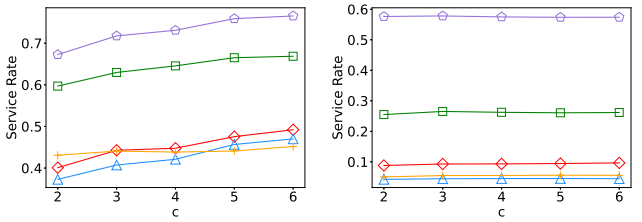


(e) Running Time (*CHD*) (f) Running Time (*NYC*)

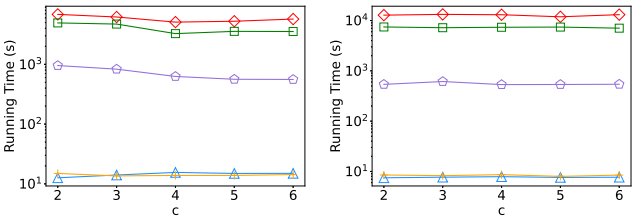
Fig. 10: Performance of Varying γ.



(a) Unified Cost (*CHD*) (b) Unified Cost (*NYC*)



(c) Service Rate (*CHD*) (d) Service Rate (*NYC*)



(e) Running Time (*CHD*) (f) Running Time (*NYC*)

Fig. 11: Performance of Varying c_r.

requests are proposed to different vehicles more decentralized in each round. Moreover, the requests only go to the next round of enumeration after being discarded. Thus, the time cost for the combination enumeration of each vehicle is significantly reduced in SARD, benefiting from our "proposal-acceptance" execution strategy. The results of RTV in *NYC* dataset with $\gamma \geq 1.8$ are not presented because the constraints of RTV-Graph exceeds the limit of *glpk* [45]. SARD performs the best in terms of unified cost, which saves up to 48.03% and 73.16% compared with other algorithms on two datasets. For the running time, SARD is $1.83\times$ to $72.68\times$ faster than RTV and GAS.

Effect of vehicle's capacity constraint. Figure 11 illustrates the results of varying the vehicle's capacity from 2 to 6. In terms of unified cost, SARD and GAS save at least 30.87% compared with other tested algorithms. The unified cost of RAND is slightly better than that of pruneGDP in the case of smaller capacity. However, with the increase of vehicle capacity, pruneGDP gradually performs better than RAND on *CHD* dataset. The reason is that the increase in vehicle capacity does not affect the random selection process of vehicles, and it has not considered the different increased costs for each vehicle. On the contrary, pruneGDP will greedily choose the vehicles with the least increased distance for service, thus pruneGDP achieves a better result with a large capacity than RAND. For a similar reason, the increase of vehicle capacity affects the number of edges between vehicles and trips in RTV-Graph, thus RTV shares a similar trend with *pruneGDP* in Fig 11(a). As for service rate, SARD is still the best among all tested algorithms ($7.58\% \sim 53.39\%$ higher service rate than other tested algorithms). In terms of running time, SARD is the fastest among batch-based methods (RTV, GAS, and SARD), which is $5.17\times \sim 17.52\times$ faster than RTV and GAS.

TABLE V: Performance of Pruning Strategies.

City	Method	Unified Cost	Service Rate	#Shortest Path Queries	Time (s)
CHD	*SARD*	263,682K	72.82%	2,971,425K	699.8
	SARD-O1	260,641K	73.34%	2,839,734K	662.0
	SARD-O2	261,969K	73.10%	2,687,618K	626.9
NYC	*SARD*	206,754K	57.64%	1,373,047K	770.6
	SARD-O1	206,470K	57.83%	1,270,116K	750.6
	SARD-O2	207,594K	57.50%	1,012,272K	595.1

Due to space limitations, please refer to Appendix A of our report [35] for the results and discussion about the experiments on varying penalty coefficient p_r and the batching time Δ.

Effects of pruning strategies. Table V shows the effect of the Ellipse and Angle pruning strategies proposed in Section III-B (parameters are in default values in Table III). We note the method without pruning strategies as *SARD*, with the Angle pruning as *SARD-O1*, with Angle and Ellipse pruning as *SARD-O2*. The effect of pruning strategies of SARD-O2 saves 9.55% and 26.28% of the shortest path queries and saves 10.42% and 22.77% of the total running time on the two datasets compared with SARD, respectively. Besides, it has no noticeable harm on the service rate and unified cost.

Memory consumption. Figure 12 shows the memory usage of tested algorithms under the default parameters. The online mode algorithms follow the first-come-first-serve mode and use less memory. In contrast, the batch mode algorithms need additional storage to store the combinations of requests in each batch (e.g., RTV-Graph for RTV, the additive index for GAS, shareability graph for SARD). Since RTV rely on an integer

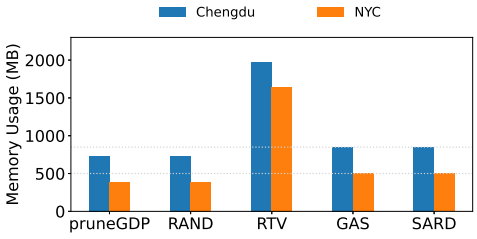


Fig. 12: Memory Consumption.

linear program, the memory usage in RTV is more than twice that in GAS and SARD. Moreover, the cost of storing an undirected and unweighted shareability graph is close to each other in SARD and GAS.

Discussion. (1) **Running Time**. The complexity of online methods, pruneGDP, and RAND, is $O(n^2)$, and thus they are the fastest in the experiments. RTV, GAS, and our SARD are batch-based algorithms. The time complexity of RTV is $O((m|G|)^c)$, where m is the number of vehicles, $|G|$ is the number of possible combinations of n tasks in the batch and can be notated as n^c. The time complexity of GAS is $O(T_s + mT_c)$, where T_s is the complexity of building additive index (as $O(n^c)$), and T_c is the time complexity of greedily searching the additive index (as O(n^c)). Our SARD needs $O(cn^2 + mn^c)$, where m and n are the numbers of vehicles and requests, respectively. For simplicity, we can note the time complexities of RTV, GAS, and SARD as $O(m^c n^2 c)$, $O(mn^c)$, and $O(mn^c)$, respectively. GAS and SARD are faster than RTV. Unlike GAS, our SARD only enumerates combinations among the proposed requests for each vehicle. Thus, n in SARD is much smaller than that in GAS. Although SARD has multiple rounds of propose-acceptance, due to the small n, it is still much faster than GAS. (2) **DataSet**. In the *NYC* dataset, the number of requests per unit time is about twice that of *CHD*. It means that in the same batch time, the algorithms based on the combination enumeration strategy (i.e., SARD and GAS) can obtain more candidate schedules, thus performing better in *NYC* than *CHD*. In addition, *NYC*'s road network is more compact than *CHD* (the nodes are only half of *CHD*), and the requests are more concentrated. Therefore, the requests in *NYC* have higher chances to share in NYC. Thus, SARD performs better in *NYC* than *CHD* in most experiments. (3) **Scalability**. For the scalability of SARD, multi-threading can speed up the building of the Shareability Graph and accelerate the acceptance stage since each vehicle makes its decision independently. In practice, our SARD can be implemented with streaming distribution computation engines (e.g., Apache Flink [46]) and apply a tumbling window strategy to extend to the distribution environment.

Summary of the experimental study:

- The batch-based methods (i.e., RTV, GAS and SARD) can get less unified costs and higher service rates than the online-based methods (i.e., pruneGDP and RAND). For instance, the service rate of SARD is up to 50% higher than that of other tested methods.
- SARD runs up to 72.68 times faster than the other batch-based methods, RTV and GAS. For example, in Figure 10(f), SARD can process *NYC* requests in 8 minutes, but GAS takes up to 9 hours.

VI. Related Work

The ridesharing problem can be reduced to a variant of the Dial-a-Ride (DARP) problem [47], [48], which aims to plan the vehicle routes and trip schedules for n riders who specified source and destination with practical constraints. The existing works on ridesharing services are categorized as static and dynamic, depending on whether all requests are known in advance. Most of the existing works [49], [50] on DARP are in static environment. For the dynamic ridesharing problem, the existing solutions are mainly in online mode [5], [25], [32], [6] or batch mode [28], [29], [51], [30].

In online mode, *insertion*[52] is the state-of-the-art operation of the existing works [53], [54] in route planning, which inserts the pickup and drop off locations of a new request into the vehicle's schedule without reordering. Tong *et al.* [25] proposed an insertion method based on dynamic programming, which checks the constraints in constant time and dispatches requests in linear time. Huang *et al.* proposed the structure of kinetic tree in [6] to trace all feasible routes for each vehicle to reduce the total drive distance. The kinetic tree always provides the optimal vehicle schedule whenever the schedule changes (i.e., a new rider arrives).

Batch-based algorithms partition the requests into groups and then assign groups to their appropriate vehicles. Alonso-Mora *et al.* [28] proposed RTV-Graph to model the relationship and constraints among requests, trips, and vehicles, where trips are the groups composed of shareable requests. The RTV-Graph minimizes the utility function by linear programming to get the allocation result between vehicles and trips. The time cost for enumerating trips in the process of building RTV-Graph grows exponentially. Zeng *et al.* [29] proposed an index called additive tree for pruning the infeasible groups during the group enumeration and greedily chose the most profitable request group for each vehicle. However, no existing work takes the priority of serving riders in the same batch into account. We proposed the structure of the shareability graph and proposed a two-phase heuristic algorithm, namely SARD, which exploits the structures to prioritize the requests.

VII. Conclusion

In this paper, we study the dynamic ridesharing problem with a batch-based processing strategy. We first proposed a graph-based shareability graph to reveal the sharing relationship between requests in each batch. With the structural information of the shareability graph, we measure the shareability loss of each request group. Furthermore, we devised a heuristic algorithm *SARD*, which adopts a two-phase strategy of "proposal-acceptance". In the experiments, the results on real datasets demonstrated that our method achieves a better service rate, less unified cost with less running time compared with the state-of-the-art batch-based methods [29], [28].

REFERENCES

[1] "[online] uberPOOL ." https://www.uber.com/, 2018.

[2] "[online] Didi Chuxing ." https://www.didiglobal.com/, 2018.

[3] B. Cici, A. Markopoulou, and N. Laoutaris, "Designing an on-line ride-sharing system," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 60:1–60:4, 2015.

[4] S. Yeung, E. Miller, and S. Madria, "A flexible real-time ridesharing system considering current road conditions," in *IEEE 17th International Conference on Mobile Data Management, MDM*, pp. 186–191, 2016.

[5] S. Ma, Y. Zheng, and O. Wolfson, "T-share: A large-scale dynamic taxi ridesharing service," in *29th IEEE International Conference on Data Engineering, ICDE 2013*, pp. 410–421, 2013.

[6] Y. Huang, F. Bastani, R. Jin, and X. S. Wang, "Large scale real-time ridesharing with service guarantee on road networks," *PVLDB*, vol. 7, no. 14, pp. 2017–2028, 2014.

[7] M. Asghari, D. Deng, C. Shahabi, U. Demiryurek, and Y. Li, "Price-aware real-time ride-sharing at scale: an auction-based approach," in *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016*, pp. 3:1–3:10, 2016.

[8] M. Asghari and C. Shahabi, "An on-line truthful and individually rational pricing mechanism for ride-sharing," in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, CA, USA, November 7-10, 2017*, pp. 7:1–7:10, 2017.

[9] D. Chu, F. Zhang, X. Lin, W. Zhang, Y. Zhang, Y. Xia, and C. Zhang, "Finding the best k in core decomposition: A time and space optimal solution," in *36th IEEE International Conference on Data Engineering, ICDE 2020*, pp. 685–696, IEEE, 2020.

[10] K. Wang, X. Cao, X. Lin, W. Zhang, and L. Qin, "Efficient computing of radius-bounded k-cores," in *34th IEEE International Conference on Data Engineering, ICDE 2018*, pp. 233–244, IEEE Computer Society, 2018.

[11] Z. Lin, F. Zhang, X. Lin, W. Zhang, and Z. Tian, "Hierarchical core maintenance on large dynamic graphs," *Proc. VLDB Endow.*, vol. 14, no. 5, pp. 757–770, 2021.

[12] Q. Linghu, F. Zhang, X. Lin, W. Zhang, and Y. Zhang, "Global reinforcement of social networks: The anchored coreness problem," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference*, pp. 2211–2226, ACM, 2020.

[13] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *International Conference on Management of Data, SIGMOD 2014* (C. E. Dyreson, F. Li, and M. T. Özsu, eds.), pp. 1311–1322, ACM, 2014.

[14] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 812–823, 2012.

[15] C. Zhang, Y. Zhang, W. Zhang, L. Qin, and J. Yang, "Efficient maximal spatial clique enumeration," in *35th IEEE International Conference on Data Engineering, ICDE 2019*, pp. 878–889, IEEE, 2019.

[16] M. Danisch, O. Balalau, and M. Sozio, "Listing k-cliques in sparse real-world graphs," in *Proceedings of the 2018 World Wide Web Conference on World Wide Web* (P. Champin, F. Gandon, M. Lalmas, and P. G. Ipeirotis, eds.), pp. 589–598, ACM, 2018.

[17] J. Cheng, Y. Ke, A. W. Fu, J. X. Yu, and L. Zhu, "Finding maximal cliques in massive networks by h*-graph," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010* (A. K. Elmagarmid and D. Agrawal, eds.), pp. 447–458, ACM, 2010.

[18] L. Chen, C. Liu, K. Liao, J. Li, and R. Zhou, "Contextual community search over large social networks," in *35th IEEE International Conference on Data Engineering, ICDE 2019*, pp. 88–99, IEEE, 2019.

[19] L. Chen, C. Liu, R. Zhou, J. Li, X. Yang, and B. Wang, "Maximum co-located community search in large scale social networks," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1233–1246, 2018.

[20] C. Giatsidis, F. D. Malliaros, D. M. Thilikos, and M. Vazirgiannis, "Corecluster: A degeneracy based graph clustering framework," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence* (C. E. Brodley and P. Stone, eds.), pp. 44–50, AAAI Press, 2014.

[21] K. Shin, T. Eliassi-Rad, and C. Faloutsos, "Corescope: Graph mining using k-core analysis—patterns, anomalies and algorithms," in *2016 IEEE 16th international conference on data mining (ICDM)*, pp. 469–478, IEEE, 2016.

[22] M. Yoon, B. Hooi, K. Shin, and C. Faloutsos, "Fast and accurate anomaly detection in dynamic graphs with a two-pronged approach," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019* (A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, eds.), pp. 647–657, ACM, 2019.

[23] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse, "Identification of influential spreaders in complex networks," *Nature physics*, vol. 6, no. 11, pp. 888–893, 2010.

[24] J. Li, X. Wang, K. Deng, X. Yang, T. Sellis, and J. X. Yu, "Most influential community search over large social networks," in *33rd IEEE International Conference on Data Engineering, ICDE 2017*, pp. 871–882, IEEE Computer Society, 2017.

[25] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu, "A unified approach to route planning for shared mobility," *PVLDB* , vol. 11, no. 11, pp. 1633–1646, 2018.

[26] P. Cheng, H. Xin, and L. Chen, "Utility-aware ridesharing on road networks," in *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1197–1210, ACM, 2017.

[27] J.-F. Cordeau and G. Laporte, "A tabu search heuristic for the static multi-vehicle dial-a-ride problem," *Transportation Research Part B: Methodological*, vol. 37, no. 6, pp. 579–594, 2003.

[28] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus, "On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment," *Proc. Natl. Acad. Sci. USA*, vol. 114, no. 3, pp. 462–467, 2017.

[29] Y. Zeng, Y. Tong, Y. Song, and L. Chen, "The simpler the better: An indexing approach for shared-route planning queries," *Proc. VLDB Endow.*, vol. 13, no. 13, pp. 3517–3530, 2020.

[30] L. Zheng, L. Chen, and J. Ye, "Order dispatch in price-aware ridesharing," *Proceedings of the VLDB Endowment*, vol. 11, no. 8, pp. 853–865, 2018.

[31] X. Bei and S. Zhang, "Algorithms for trip-vehicle assignment in ride-sharing," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)* (S. A. McIlraith and K. Q. Weinberger, eds.), pp. 3–9, AAAI Press, 2018.

[32] Y. Xu, Y. Tong, Y. Shi, Q. Tao, K. Xu, and W. Li, "An efficient insertion operator in dynamic ridesharing services," in *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pp. 1022–1033, IEEE, 2019.

[33] J. Pan, G. Li, and J. Hu, "Ridesharing: Simulator, benchmark, and evaluation," *Proc. VLDB Endow.*, vol. 12, no. 10, pp. 1085–1098, 2019.

[34] D. O. Santos and E. C. Xavier, "Dynamic taxi and ridesharing: A framework and heuristics for the optimization problem," in *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013* (F. Rossi, ed.), pp. 2885–2891, IJCAI/AAAI, 2013.

[35] "[online] Technical Report ." https://cspcheng.github.io/pdf/ShareGraphRidesharing-Report.pdf, 2021.

[36] S. Ma, Y. Zheng, and O. Wolfson, "Real-time city-scale taxi ridesharing," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1782–1795, 2015.

[37] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos, "Community detection in social media - performance and application considerations," *Data Min. Knowl. Discov.*, vol. 24, no. 3, pp. 515–554, 2012.

[38] O. Rokhlenko, Y. Wexler, and Z. Yakhini, "Similarities and differences of gene expression in yeast stress conditions," *Bioinform.*, vol. 23, no. 2, pp. 184–190, 2007.

[39] R. M. Karp, "Reducibility among combinatorial problems," in *Proceedings of a symposium on the Complexity of Computer Computations* (R. E.

Miller and J. W. Thatcher, eds.), The IBM Research Symposia Series, pp. 85–103, Plenum Press, New York, 1972.

[40] J. Bhasker and T. Samad, "The clique-partitioning problem," *Computers & Mathematics with Applications*, vol. 22, no. 6, pp. 1–11, 1991.

[41] S. Janson, T. Łuczak, and I. Norros, "Large cliques in a power-law random graph," *Journal of Applied Probability*, vol. 47, no. 4, pp. 1124–1135, 2010.

[42] Y. Li, L. H. U, M. L. Yiu, and N. M. Kou, "An experimental study on hub labeling based shortest path algorithms," *Proc. VLDB Endow.*, vol. 11, no. 4, pp. 445–457, 2017.

[43] Y. Huang, F. Bastani, R. Jin, and X. S. Wang, "Large scale real-time ridesharing with service guarantee on road networks," *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 2017–2028, 2014.

[44] J. Wang, P. Cheng, L. Zheng, C. Feng, L. Chen, X. Lin, and Z. Wang, "Demand-aware route planning for shared mobility services," *Proc. VLDB Endow.*, vol. 13, no. 7, pp. 979–991, 2020.

[45] "[online] GNU GLPK ." https://www.gnu.org/software/glpk/, 2016.

[46] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.

[47] N. H. Wilson, R. Weissberg, B. Higonnet, and J. Hauser, "Advanced dial-a-ride algorithms," tech. rep., 1975.

[48] J.-F. Cordeau and G. Laporte, "The dial-a-ride problem (darp): Variants, modeling issues and algorithms," *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, vol. 1, no. 2, pp. 89–101, 2003.

[49] K.-I. Wong and M. G. Bell, "Solution of the dial-a-ride problem with multi-dimensional capacity constraints," *International Transactions in Operational Research*, vol. 13, no. 3, pp. 195–208, 2006.

[50] J.-F. Cordeau, "A branch-and-cut algorithm for the dial-a-ride problem," *Operations Research*, vol. 54, no. 3, pp. 573–586, 2006.

[51] L. Zheng, P. Cheng, and L. Chen, "Auction-based order dispatch and pricing in ridesharing," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1034–1045, IEEE, 2019.

[52] J.-J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. Wilson, "A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows," *Transportation Research Part B: Methodological*, vol. 20, no. 3, pp. 243–257, 1986.

[53] I. Ioachim, J. Desrosiers, Y. Dumas, M. M. Solomon, and D. Villeneuve, "A request clustering algorithm for door-to-door handicapped transportation," *Transportation science*, vol. 29, no. 1, pp. 63–78, 1995.

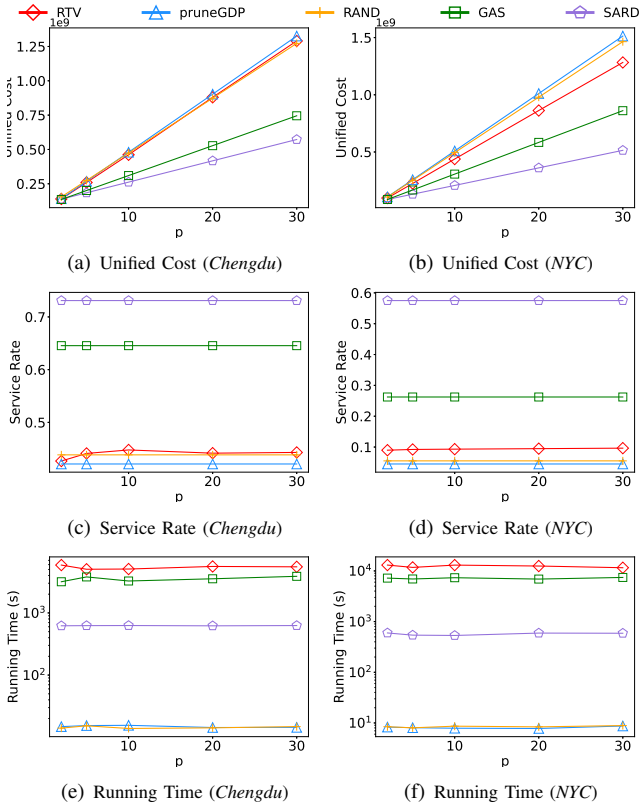
[54] L. Häme, "An adaptive insertion algorithm for the single-vehicle dial-a-ride problem with narrow time windows," *European Journal of Operational Research*, vol. 209, no. 1, pp. 11–22, 2011.

APPENDIX

A. *Effects of penalty and batching period.*

Effect of penalty. Figure 13 represents the results of varying the penalty coefficient from 2 to 30. The experimental results show that the service rate of most methods does not influence by the varying penalty. pruneGDP, GAS and SARD, mainly take the increased distance, the maximum group profit, and the shareability loss as their indicators in greedy strategies in the assignment phase, but the penalty coefficient will only affect them. However, RTV utilized the penalty coefficient in the constraint matrix of linear programming (LP), which will have a specific effect on the result of LP when the penalty coefficient is small. Still, the varying penalty coefficient on RTV can be ignored when the penalty coefficient is large enough. In terms of unified cost, each experiment algorithm is proportional to the penalty coefficient. SARD still takes the lead in two datasets, with service rate increased by $8.55\% \sim 52.99\%$. Similarly, since the penalty coefficient does not affect the assignment phase, the execution time does not change obviously on the two datasets.

Effect of the batching period. Figure 14 represents the results of varying the batching period from 1 to 9 seconds for batch-based methods. The varying batch time affects the batch-based algorithms in terms of unified cost and service rate. SARD performs the best with the varying the batching period, which saves up to 44.18% and 53.08% in unified cost and increases the service rate by up to 29.2% and 48.96% in two datasets, respectively. For the running time, since the number of execution rounds will decrease when the period of each batch increases, the running times of these methods decrease. SARD is $5.17\times \sim 24.47\times$ faster than RTV and GAS.

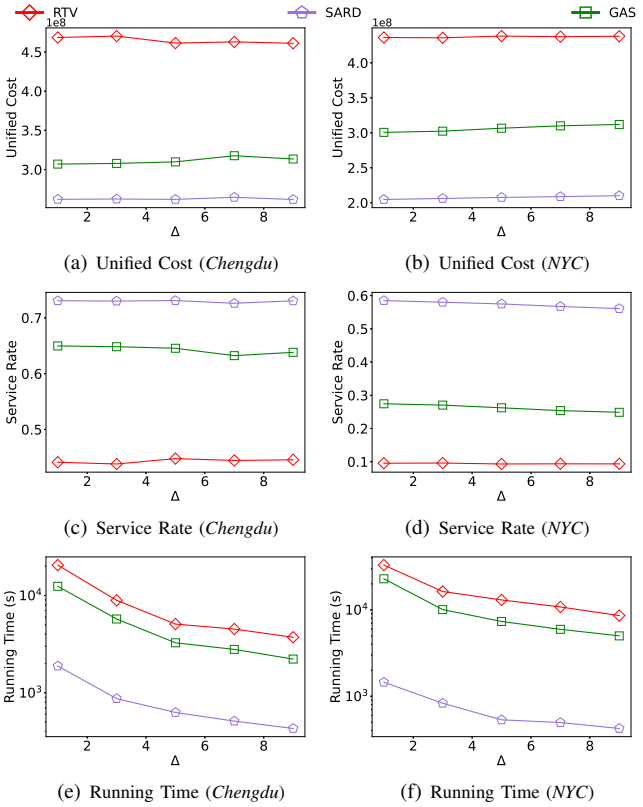


(a) Unified Cost (*Chengdu*) (b) Unified Cost (*NYC*)

(c) Service Rate (*Chengdu*) (d) Service Rate (*NYC*)

(e) Running Time (*Chengdu*) (f) Running Time (*NYC*)

Fig. 13: Performance of varying p_r.



(a) Unified Cost (*Chengdu*) (b) Unified Cost (*NYC*)

(c) Service Rate (*Chengdu*) (d) Service Rate (*NYC*)

(e) Running Time (*Chengdu*) (f) Running Time (*NYC*)

Fig. 14: Performance of varying Δ.