

# STRide: a Framework to Exploit the Structure Information of Shareability Graph in Ridesharing Services

Yu Chen  
East China Normal University  
Shanghai, China  
yu.chen@stu.ecnu.edu.cn

Lei Chen  
The Hong Kong University of Science and Technology  
Hong Kong SAR, China  
leichen@cse.ust.hk

Peng Cheng  
East China Normal University  
Shanghai, China  
pcheng@sei.ecnu.edu.cn

Wenjie Zhang  
The University of New South Wales  
Sydney, Australia  
wenjie.zhang@unsw.edu.au

## ABSTRACT

Ridesharing services play an essential role in modern transportation, which significantly reduces traffic congestion and exhaust pollution. In ridesharing problem, improving the sharing rate between riders can not only save the travel cost of riders and drivers, but also utilize of 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, graph is a powerful tool to analyze the structure information between nodes. Therefore, in this paper, we propose 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 problem 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.

## PVLDB Reference Format:

Yu Chen, Peng Cheng, Lei Chen, and Wenjie Zhang. STRide: a Framework to Exploit the Structure Information of Shareability Graph in Ridesharing Services. PVLDB, 15(1): XXX-XXX, 2022.  
doi:XX.XX/XXX.XX

## PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://anonymous.4open.science/r/STRide-SARD/>.

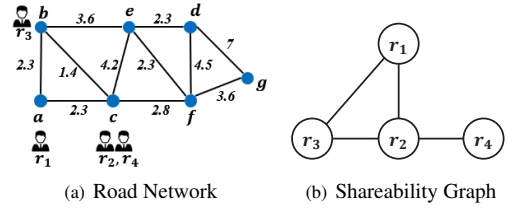
## 1 INTRODUCTION

Recently, ridesharing has become a popular public transportation choice, which makes a great contribution to reduce energy and relieve traffic pressure. In ridesharing, a driver can serve different

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.  
Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.  
doi:XX.XX/XXX.XX

**Table 1: 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



**Figure 1: An Motivation Example**

riders simultaneously once riders can share parts of their trips and there are enough seats for them. The ridesharing service providers (e.g., Uber [3], Didi [2]) are constantly pursuing better service quality, such as higher servicing rate [16, 50], lower total travel distance [24, 36], or higher total revenue [5, 6].

For ridesharing platforms, *vehicle-request matching* and *route planning* are two critical issues to address. Given 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 two requests can be served by any vehicle 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 [15, 34, 35, 46],  $k$ -truss [23, 44], clique [12, 20, 53]) and contribute to specific applications (e.g., community discovery [10, 11, 21], anomaly detection [42, 51], influential analysis [30, 31], etc). However, to the best of our knowledge, no existing works have targeted on utilizing the *structure properties* of the shareability graph to guide the vehicle-request matching to improve the service quality.

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

**Example 1.** In a ridesharing example shown in Figure 1(a), there are four requests  $r_1, r_2, r_3, r_4$  and 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 detailed information of requests is shown in Table 1, where the release time represents the time when the request is submitted to the platform, and deadline indicates the latest time to reach the destination. Suppose that the platform only allows a vehicle to serve at most 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 request  $r_4$  can only share with  $r_2$ . We can construct a shareability graph of the four requests as shown in Figure 1(b), where each edge indicates the connected two requests can share with each other.

If the platform uses an online assignment framework [24, 36, 43] for allocation (i.e., each request will be served or discarded immediately when it arrives), the platform 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  $r_1, r_2, r_3, r_4$  are 2, 3, 2, 1, respectively. In a shareability graph of requests, 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  $r_3$ .

Therefore, the requests could be associated with different priorities to improve the service rate of the platform, which with higher priority will be arranged firstly to achieve a better objective.

In real platforms, vehicles and requests often arrive dynamically. Existing works on ridesharing can be summarized into two modes: the *online mode* [13, 19, 24, 36] and *batch mode* [4, 7, 52, 54]. The state-of-the-art operator for the online mode is *insertion* [13, 19, 43, 49, 54], which inserts the source and destination of a request into proper positions of the vehicle’s route without reordering its current trip schedule such that the increase of the total travel cost is minimized. Batch-based methods [4, 38, 41, 52, 54] 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 relative 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 [4]). Can we have a more efficient batch-based approach to have better results for a relatively long time period (e.g., one day)?

In this paper, we propose 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 algorithm with graph analyses of the shareability graph in ridesharing problems. In our STRide framework, we first propose a fast shareability graph builder, which efficiently

**Table 2: 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

extracts shareable relationships between requests through ellipse and angle-based filtering to only maintain the good shareable relationships (i.e., requests with close travel directions such that the detours of the serving vehicles are intuitively small). To exploit the structure information of shareability graph, we devised an efficient algorithm, namely *structure-aware ridesharing dispatch* (SARD), to take the cohesiveness of requests in shareability graph (evaluated through the structure information of the graph) into consideration and revise the priority of each request through analyzing its shareability in the assignment phase. SARD adopts a two-phase strategy of “proposal-acceptance”, 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 method. In this way, the batch-based method can be more efficient in large-scale ridesharing problem.

To summarize, we make the following contributions in this paper:

- We introduce our STRide framework, which handles incoming requests in batch mode and adjusts the matching priority of requests in each batch in Section 2.3.
- We propose a graph structure called *shareability graph* for analyzing the sharing relationships between requests intuitively ridesharing problem, and designed an efficient algorithm to generate the shareability graph for each batch in Section 3.
- 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 4.
- We conduct extensive experiments on two real datasets to show the efficiency and effectiveness of our proposed STRide framework in Section 5.

In addition, the remaining sections of the paper are arranged as follows. We define the shareability graph and propose an efficient construction algorithm in Section 3. We review and compare previous studies on queuing theory and vehicle dispatching in Section 6 and conclude the work in Section 7.

## 2 PROBLEM DEFINITION

In this section, we introduce and analyze the batch-based dynamic ridesharing problem *BDRP* studied in this paper. Then, we introduce the overview of our STRide framework.

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  $v$ . When the speed is known, the minimum travel time cost can be directly transferred to the shortest travel distance.

## 2.1 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 arriving destination before time  $d_i$ .

For each *available* ridesharing request  $r_i$ , there implies a constraint of the maximum waiting time  $w_i = d_i - \text{cost}(s_i, e_i)$ , which can be used to prune vehicles that cannot arrive at  $s_i$  within  $w_i$ . Meanwhile, the deadline of  $r_i$  demands that the assigned vehicle must deliver the riders to the destination  $e_i$  before the 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 shows.

**Definition 2 (Schedule).** Given a vehicle  $v_j$  with  $m$  allocated requests  $R_j = \{r_1, \dots, r_m\}$ . Let  $S_j = \langle o_1, \dots, o_{2m} \rangle$  defines 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 *way point*. 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.

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

Here,  $\text{ddl}(o_k)$  is the deadline  $d_i$  when  $o_k$  is the destination of  $r_i$ , while  $\text{ddl}(o_k) = d_i - \text{cost}(s_i, e_i)$  when  $o_k$  is the source of  $r_i$ .

To complete the arranged schedule on time, the vehicle has a maximum allowed detour distance at each way point. We define the maximum allowed detour time as *buffer time* in Definition 3.

**Definition 3 (Buffer Time [14, 43]).** Given a valid vehicle schedule  $S_i = \langle o_1, o_2, \dots, o_{2m} \rangle$ , let  $\text{buf}(o_x)$  be the maximum detour time at the way point  $o_x$  without violating the deadline constraints of its consequent way points.

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

where  $\text{arrive}(o_{x+1})$  is the earliest arriving time of way point  $o_{x+1}$  without any detour in previous way points in schedule  $S_i$ .

With the buffer time of each way point, we can find a suitable pickup and drop-off time for the new coming request in  $O(n)$  time [43].

Based on these definitions above, *RTV-Graph* is proposed in [4] to describe the shareable relationship between requests and vehicles. The solution proposed for the dynamic ridesharing problem which based on RTV graph and linear programming in [4] could get a better result with the shorter total distance and higher service rate. However, in the process of constructing RTV graph, it needs to check whether any two or more requests can be shared with  $O(n^c)$  time, and check whether each car can serve all requests in each trip in  $O(m|T|)$  time. Therefore, it's not efficient enough to be used in a large-scale real application. In this paper, we proposed a simpler shareability graph for analyzing the shareable relationships instinctively to help solve dynamic ridesharing problem defined as following.

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

In *Batched Dynamic Ridesharing Problem (BDRP)*, we need to resolve the 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  $\mathbb{G} = \{G_1, G_2, \dots, G_z\}$ , which satisfied 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 for each vehicle while minimizing the utility function. In this work, we refer to the unified cost function  $UC$  in [43] and defined the following utility function  $U$ :

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

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

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

Table 2 summarizes the commonly used symbols.

## 2.2 Hardness of Batched Dynamic Ridesharing Problem

**Theorem 2.1 (Hardness of the BDRP).** *The Batched Dynamic Ridesharing Problem defined in Definition 4 is NP-hard.*

**PROOF.** We prove the theorem by a reduction from the URR problem defined in [14], which has been proved to be a NP-hard problem. The URR problem can be brief described as following: Given a set  $R$  of  $m$  riders and a set  $V$  of  $n$  vehicles, each rider is associated with a source location  $s_i$ , destination location  $e_i$ , a pickup deadline  $rt_i^-$  and a drop-off deadline  $rt_i^+$ . The URR problem arranges riders to vehicles to maximize the utility function  $u$  with capacity constraint and time constraint.

For a given URR problem, we can transform it to an instance of BDRP problem: we partition the riders  $r_i \in R$  into a single element set  $G_i$ , the route of  $G_i$  is a simple shortest path from  $s_i$  to  $e_i$ . In addition, we set the utility value for each vehicle and group pairs as  $\mu'(v_j, G_{v_j}) = -\mu(v_j, r_i)$ . Then, for this BDRP instance, we would like to arrange a request group for the given vehicle with a route such that the summation utility value  $\sum_{v_j \in V} -\mu'(v_j, G_{v_j})$  is minimized. This shows that the URR problem can be solved if and only if the transformed BDRP can be solved.

In this way, we can reduce the URR problem to the BDRP. Since the URR problem has been proved to be NP-hard, BDRP is also NP-hard. This completes the proof of the theorem.  $\square$

Since BDRP is NP-hard, it is intractable. To efficiently and effectively solve BDRP, we propose a novel batch based framework, namely STRide, to exploit the structure information of shareability graph of requests to improve the performance of the dynamic ridesharing system on the service rate and total travel cost.

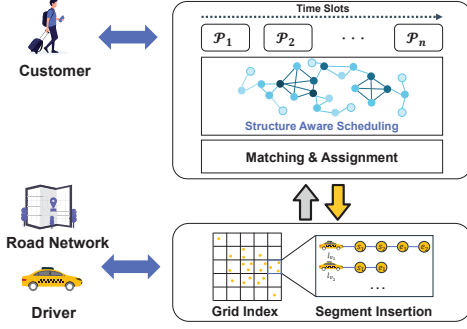


Figure 2: Overview of System Framework STRide.

### 2.3 An Overview of STRide Framework

We first briefly introduce the major parts of our STRide framework, which is illustrated in Figure 2.

**Index Structure.** In ridesharing, vehicles keep moving over time, which requires that the index we used must update efficiently. Grid index is an efficient index structure that can query and update moved vehicles in constant time (i.e., more efficient than many other spatial indexes such as tree-based indexes [33]). 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 widely used solutions: *kinetic tree insertion* [24] and *linear insertion* [43, 49]. Kinetic tree insertion maintains all feasible schedules for a particular vehicle in a tree structure, and always returns the optimal schedule (i.e., the shortest total traveling cost). In contrast, the linear insertion method only maintain a current optimal schedule and inserts the way points into the maintained schedule in constant time without *reordering* way points, which means that it may miss the optimal schedule. However, Ma *et al.* [37] reveals that reordering way points 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 3 to analyze the sharing relationship between requests. In the *Matching and Assignment* phase, we proposed a two-phase algorithm SARD in Section 4, where the requests propose to valid vehicles in descending order of the additional travel cost (i.e., propose to vehicles needing more additional travel costs first), and vehicles select a group of structural friendly requests to accept. Here, we propose a novel measurement, namely shareability loss, to evaluate the harmfulness on merging a group of requests to a super node with respect to the sharing probability of the rest nodes in the shareability graph. The discarded requests will propose to other vehicles in the next proposal phase to improve the service rate.

## 3 SHAREABILITY GRAPH

Although the requests in a same batch have similar release times, the sharing probabilities of them can vary dramatically. For example, a request with a pair of popular source and destination has more opportunities to share a vehicle with other requests. Thus, an intuitive motivation for optimizing the grouping process is to associate

higher priorities to the requests with lower shareability. Therefore, in this section, we proposed a *shareability graph* to manage the sharing relationships between requests with an efficient construction algorithm for it. With shareability graph, we can further analyze the shareability of requests in each batch.

### 3.1 Shareability Graph

Given a pair of requests  $r_a$  and  $r_b$ , we called  $r_a$  and  $r_b$  are *shareable* if there exist at least one feasible route for  $r_a$  and  $r_b$  to share.

**Definition 5 (Shareability Graph).** Given a set of  $m$  requests  $R$ , let  $SG = \langle R, E \rangle$  denotes a *shareability graph* with a set of nodes  $R$  and edges  $E$ , where each node in  $R$  indicates a request. Each 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  $\langle s_1, s_2, e_2, e_1 \rangle$ ). With the shareability graph, we can easily find out the candidate shareable requests of each request by retrieving the neighbors in the shareability graph. Moreover, we have the following two observations with graph analysis techniques.

**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 through comparing their degrees in the shareability graph. We call the degree of a request as its *shareability*. For example, a request with a small shareability is often more urgent to select an appropriate shareable request; otherwise it will not share with any requests in the batch.

**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 [12, 39, 40]. In the shareability graph, by Theorem 3.1, 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.

**Theorem 3.1.** *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. We have a set,  $R$ , of  $k$  requests, and there exist a valid sharing schedule  $S$  consisting of their sources and destinations. Assume the corresponding nodes of  $k$  requests in the shareability graph do not form a  $k$ -clique, which means there are at least two requests,  $r_a, r_b \in R$ , not connected.

Let  $S'$  be a subsequence of  $S$  through removing the sources and destinations of all other requests except for  $r_a$  and  $r_b$ . Removing the way points from  $S$  will reduce the detours, which will maintain the validation of  $S$ . Thus,  $S'$  must still be a valid schedule. According to the definition of shareability graph, there must be an edge connecting  $r_a$  and  $r_b$  in the corresponding shareability graph, which makes a contradiction with our assumption. In conclusion, the nodes of  $k$  requests forms a  $k$ -clique in the shareability graph.  $\square$

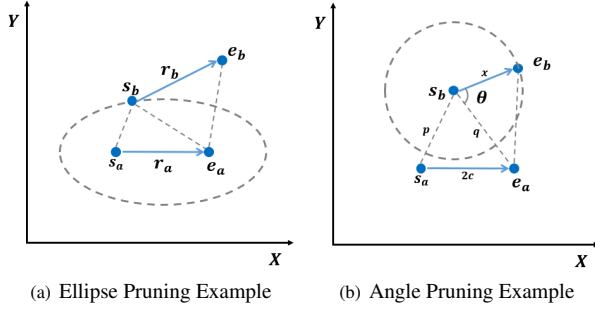


Figure 3: A Illustration Example for Shareability Graph Builder

### 3.2 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 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 propose two pruning strategies, ellipse pruning and angle pruning illustrated in Figure 3, for more efficient shareability graph construction. Note that, when we prune the candidates 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 way point.

**Ellipse pruning.** Our first 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. Therefore, for each request  $r_a$ , we prune the requests whose sources locating out the ellipse of  $r_a$  defined in Theorem 3.2 to avoid unnecessary shortest path query.

**Theorem 3.2.** 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  $\text{cost}(s_a, s_b) + \text{cost}(s_b, e_a) \leq d_a$ .

**PROOF.** For any candidate shareable request  $r_b$ , there are only two feasible paths 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 denoted as  $\text{arrive}(e_a) = \text{cost}(s_a, s_b) + \text{cost}(s_b, e_a)$ . Recall that  $r_a$  must be delivered to destination before deadline  $d_a$ , thus,  $\text{arrive}(e_a) = \text{cost}(s_a, s_b) + \text{cost}(s_b, e_a) \leq d_a$ . This completes the proof.  $\square$

**Angle pruning.** Our second pruning strategy is based on the travel directions of the requests, as we notice that the requests with similar travel directions have a higher probability to share the trips. For instance, a northward request is hard to share with a southward request, because the driver needs to turn around and spend a lot of time to drive in the opposite direction to deliver the other passenger after finished one of them. We defined a threshold  $\delta$  to prune request pairs that have similar sources but divergent directions with Theorem 3.3. Let  $\vec{ab}$  be the vector pointing from  $a$  to  $b$ .

**Theorem 3.3.** For a request  $r_a$  and its candidate sharing request  $r_b$ , the expected probability of successfully sharing a trip will increase when the angle  $\theta$  between  $\vec{s_b e_a}$  and  $\vec{s_b e_b}$  decreases.

#### Algorithm 1: Shareability Graph Builder

**Input:** A set  $R$  of  $n$  requests with an angle threshold  $\delta$

**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 3.2
4   for  $r_b \in C$  do
5     if  $\arccos\left(\frac{\vec{s_b e_a} \cdot \vec{s_b e_b}}{|\vec{s_b e_a}| |\vec{s_b e_b}|}\right) \in \left[-\frac{\delta}{2}, \frac{\delta}{2}\right]$  then
6       if  $r_a$  and  $r_b$  are shareable then
7          $E = E \cup (r_a, r_b)$ 
8 return  $SG = \langle V, E \rangle$ 

```

**PROOF.** Firstly, we denoted the maximum detour time for request  $r_a$  as  $(\alpha - 1) \times \text{cost}(s_a, e_a)$  where  $\alpha \geq 1$ . 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)  $p + x + \text{cost}(e_a, e_b) \leq 2\alpha c$ ; (b)  $p + q + \text{cost}(e_a, e_b) \leq \alpha x$ , where  $p, q, x$  and  $2c$  indicate  $\text{cost}(s_a, s_b)$ ,  $\text{cost}(s_b, e_a)$ ,  $\text{cost}(s_b, e_b)$  and  $\text{cost}(s_a, e_a)$ , respectively.

When request  $r_a$ , source  $s_b$  and  $\text{cost}(s_b, e_b)$  are fixed, the total travel cost of the two possible schedules only depends on  $\text{cost}(e_a, e_b)$ . Then,  $\text{cost}(e_a, e_b)$  can be represented through  $x, n$  and  $\cos(\theta)$ . In (a), we have  $x \leq \frac{(2\alpha c - p)^2 - q^2}{4\alpha c - 2p - 2q \cos(\theta)} \leq 1 / \left( \frac{\cos^2(\theta/2)}{(\alpha+1)c} + \frac{\sin^2(\theta/2)}{(\alpha-1)c} \right)$  (noted as  $g(c)$ ), because  $p + q \leq 2(\alpha+1)c$  and  $p - q \leq 2(\alpha-1)c$  hold in the ellipse in Theorem 3.2. In (b), we have  $\alpha x \geq p + q + \sqrt{(x - q + \cos(\theta))^2}$ , which can be rewritten as  $x \geq \frac{2c(1 - \cos(\theta))}{\alpha - 1}$  (marked as  $h(c)$ ) by  $p + q \geq 2c$ . It is worth noting that in both cases, with the decrease of the angle, the possible range of  $x$  increases gradually. In other words, for a given candidate request  $r_b$ , when the angle  $\theta$  between  $\vec{s_b e_a}$  and  $\vec{s_b e_b}$  decreases, the travel cost  $x = \text{cost}(s_b, e_b)$  is more likely to satisfy the deadline constraint.  $\square$

By investigating the real datasets of Chengdu and New York City in Section 5, 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  $\mu$  and  $\sigma$  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  $\theta$  is greater than a given threshold  $\delta$  can be evaluated as follows:

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

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

### 3.3 Shareability Graph Builder

The details of shareability graph builder algorithm is 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 shareable relationship for each node successively (lines 2~7). With the benefit of Theorem 3.2, 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  $\vec{s_b e_b}$  by the source and destination of the corresponding request to represent the distance and direction. If the angle  $\theta$  of  $\vec{s_b e_a}$  and  $\vec{s_b e_b}$  is out of the given angle threshold  $\delta$ ,  $r_b$  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.** Suppose that the shortest path query can be finished in  $O(q)$ . 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. In addition, the angle calculation can be also completed in constant time. 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  $O(q \cdot n^2)$ .

## 4 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 under the guidance of shareability graph. Specifically, we first discuss two main schedule maintenance methods in Section 4.1. Then we introduce a bottom-up enumeration strategy for different combination of requests in Section 4.2. Lastly, we elaborate the detail steps of SARD in Section 4.3.

### 4.1 Schedule Maintenance

There are two state-of-art strategies for schedule maintenance in previous work: kinetic tree insertion [24] and linear insertion [43, 49]. Kinetic tree maintains all feasible schedules and check all available way points ordering exhaustively to insert a new request. Even kinetic tree can get the *exact* optimal schedule for the vehicle, it needs to maintain up to  $\frac{(2m)!}{2^m}$  schedules in the worst case ( $m$  is the number of requests assigned in vehicle). In contrast, the schedule obtained by 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 5, 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 global optimal schedule is up to 89% and 85%, when we insert the third and forth request, respectively. To improve the probability of achieving global optimal schedule with the linear insertion method, we reorder the insertion sequence of the requests base on the Observation 1 in Section 3.1. Specifically, we first select two requests with the lowest shareability (i.e., the degree of node) in 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%.

### 4.2 Request Grouping

In batch mode, we need to enumerate all feasible request combinations before the assignment phase. If we simply construct groups by enumerate all possible combinations, we need to check  $\sum_{i=1}^c \binom{n}{i}$  groups, where  $c$  is the capacity of vehicles. For each group, we need

---

### 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  $\{RG_1, RG_2, \dots, RG_c\}$  as empty sets
2 forall  $r_a \in R$  do
3    $RG_1.insert(\{r_a\}, \langle s_a, e_a \rangle)$ 
4 for  $l \in [2..c]$  do
5   foreach pair  $(G_x, G_y)$  in  $RG_{l-1}$  do
6      $G \leftarrow G_x \cup G_y$ 
7     if  $|G| = l$  and  $G$  satisfied Lemma 4.1 then
8        $r_b \leftarrow$  find maximum degree node in  $SG$ 
9        $S \leftarrow$  insert  $r_b$  into the schedule  $S'$  of  $G \setminus \{r_b\}$ 
        maintained in level  $RG_{l-1}$ 
10      if  $S'$  is valid then
11         $RG_l.insert(G, S)$ 
12 return  $\mathcal{RG} = \{RG_1, RG_2, \dots, RG_c\}$ 

```

---

to verify that whether there exists a feasible route to serve these requests in  $O(k^2)$  by linear insertion [43], where  $k$  is the size of group. However, many groups are invalid for vehicles in practice. To avoid unnecessary enumeration, Zeng *et al.* propose an index called *additive tree* [52], which enumerates valid groups level by level through a tree-based structure. In 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 to prune some invalid groups in the enumeration, it still needs to maintain all valid schedules for each node of the additive tree. 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 4.1, we can heuristically reorder the insertion sequence to get an optimal schedule with higher probability. Therefore, 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  $\{RG_1, RG_2, \dots, RG_c\}$  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 is consisted of its source and destination for level 1 of the modified additive tree (line 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  $RG_{l-1}$  (line 4~11). During constructing the nodes in level  $l$ , we only consider the groups  $G$  with  $l$  requests and  $G$  must satisfy Lemma 4.1 (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 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  $RG_l$  (line 10~11).

**Lemma 4.1.** For any valid group  $G_x$ : (a)  $\forall r_a \in G_x$ , the group  $G_x \setminus \{r_a\}$  must be also valid; and (b) the nodes of  $r_a \in G_x$  forms a clique in the shareability graph.



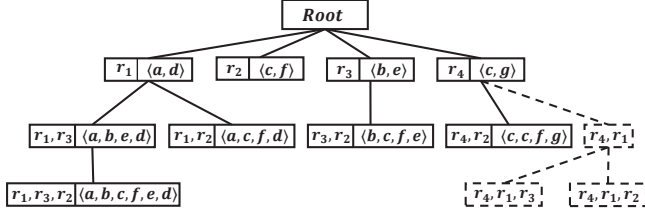


Figure 4: Grouping Tree in Example 2.

PROOF. The first condition has been proved by Lemma 2 in [52] and condition (b) is derived from Theorem 3.1.  $\square$

**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 as shown 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  $O(n^c)$ . For each new group, we only need  $O(c)$  time to perform linear insert operation for a new schedule. We can finish the request grouping in  $O(c \cdot n^c)$ .

### 4.3 SARD Algorithm

With the grouping algorithm in Section 4.2, 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 beginning propose 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 can propose to other better vehicles in the next round. 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 super node  $\hat{v}_i$ . After we substitute a super node  $\hat{v}_i$  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  $S\text{Loss}(G_i)$  of substituting a super-node

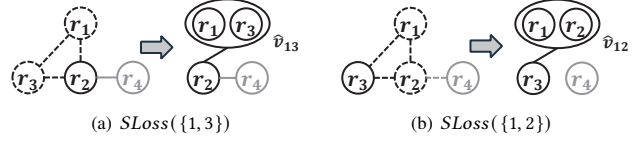


Figure 5: An Illustration Example of Shareability Loss

$\hat{v}_i$  for a  $k$ -clique group  $G_i \subseteq V$  is evaluated with the following structure-aware loss function:

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

where  $N(v)$  is the set of neighbor nodes of  $v$  in the shareability graph SG.  $S\text{Loss}(G) = \deg(r)$  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 super node 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 super node  $\hat{v}_{13} = \{r_1, r_3\}$  by Theorem 3.1. 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  $S\text{Loss}(\{r_1, r_3\}) = 3 - 1 = 2$ . In Figure 5(b), we try to merge  $r_1$  and  $r_2$  into a super node  $\hat{v}_{12} = \{r_1, r_2\}$ . 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  $S\text{Loss}(\{r_1, r_2\}) = 4 - 1 = 3$  here. Therefore, substituting  $\{r_1, r_3\}$  is more structure-friendly than substituting  $\{r_1, r_2\}$ .

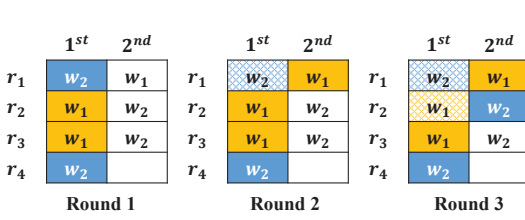
Shareability loss can guide a vehicle to select a set of requests to serve from the potential requests (i.e., proposed to the vehicle in the proposal phase). Specifically, a vehicle  $v_j$  should select a set of requests whose shareability loss is the minimum among all groups of its potential requests. In Theorem 4.1, we prove that through serving a group of requests with the smallest shareability loss, the remaining requests can have higher upper bound of sharing rate.

**Theorem 4.1.** Substituting a super-node for group  $G$  with smaller shareability loss will increase the upper bound of sharing probabilities more for the remaining nodes in the shareability graph.

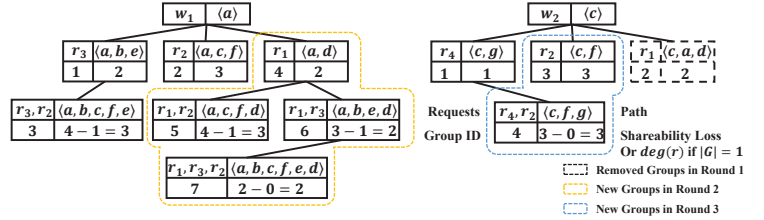
PROOF. According to Theorem 3.1, all valid groups we picked in the assignment phase always constitute a  $k$ -clique in the shareability graph. Therefore, we model the problem of maximizing sharing probability as a problem of clique partition [29], which trying to find the minimum number of cliques to cover the graph such that each node appear in exactly one clique. In the worst case, we will partition each node in the shareability graph into a 1-clique, and then the sharing probability of the nodes is 0 (i.e., no requests can share with each other). Intuitively, the less clique we used to cover the shareability graph, the higher the sharing probability is. Thus, we transform our problem into clique partition. Moreover, because of the capacity constraint  $k$  to each feasible group in our problem, we consider the problem of partitioning the shareability graph into minimum number of cliques which size is not larger than  $k$ . In [8], 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, in line with the characteristics



(a) Transformation of the Candidate Queue



(b) Grouping Trees for Vehicles

Figure 6: An Example for the SARD Algorithm

of power-law distribution. For analytical tractability, we assume that the degree of the node in the shareability graph as a random variable  $\delta$  following the power-law distribution with exponent  $\alpha$ , which complementary cumulative distribution shown as  $\Pr(\delta \geq x) = ax^{-\alpha}$ . For a shareability graph  $SG$  having  $n$  nodes and the degree of its each node follows the power-law distribution with exponent  $\alpha$ , Janson *et al.* [27] reveal that the size of the largest clique  $\omega(SG(n, \alpha))$  in graph  $SG$  is a constant with  $\alpha > 2$ . However, in the heavy-tail distribution,  $\omega(SG(n, \alpha))$  grows with  $n^{1-\alpha/2}$  (the formal description is shown in equation 7).

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

For any shareability graph  $SG$ , we deal with the optimal partition  $\{C_1 \dots, C_\theta\}$  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  $\theta'_{upper}$  after scaling of our problem with  $n$  and  $e$  by formula 6 and 7.

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

The higher the number of edges  $e$  in the shareability graph  $SG$  is, the lower the upper bound  $\theta'_{upper}$  of the number of clique partitions is. In conclusion, substituting a super node for the group of nodes with the lowest edge loss will keep the most number of edges left in the shareability graph  $SG$ , which improves the sharing probabilities of the remaining nodes in  $SG$ . This completes the proof.  $\square$

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

**Theorem 4.2.** *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 minimum clique partitions with size no larger than  $k$  on  $SG$  are  $\{C_1, \dots, C_n\}$ , and  $v_x$  and  $v_y$  are included in  $C_a$  and  $C_b$ , respectively. The size of the clique  $|C_a| \leq 2$  because  $\deg(v_x) = 1$ . If  $v_x$  and  $v_y$  are included in different partitions ( $a \neq b$ ), the size of the clique  $|C_a| = 1$  because  $v_x$  is not connected to any node except  $v_y$ . Thus, removing  $v_y$  from  $C_b$  and merging it into  $C_a$  will not increase the number of isolated groups, whose sizes are not larger 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.  $\square$

With Theorems 4.1 and 4.2, 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

### 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  $W \leftarrow$  initialize a working set 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      $Q_{r_a} \leftarrow$  initialize a priority queue by  $\Delta$  utility
- 7     insert candidate vehicles into  $Q_{r_a}$
- 8     **while**  $\exists r_a \in R_p, |Q_{r_a}| \neq 0$  **do**
- 9       **foreach**  $r_a \in R_p$  **and**  $|Q_{r_a}| \neq 0$  **do**  $\triangleright$  Proposal Phase
- 10           $w_x \leftarrow Q_{r_a}.\text{pop}()$  the worst vehicle
- 11           $R_{w_x}.\text{insert}(r_a)$
- 12          **foreach**  $w_x \in W$  **do**  $\triangleright$  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**  $W$

timestamps. When each batch arrives, we insert the requests in this batch  $R_p$  (line 3) and process them together with the available (unmatched and unexpired) request 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  $Q_{r_a}$  in descending order of the increased travel cost for serving  $r_a$  (lines 5~7). In the proposal phase (lines 9~11), each request which has not been accepted in the previous round will propose to its current worst vehicle (i.e., the vehicle with the largest travel cost increase on serving the request) (line 11). After that, we enumerate 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 the 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 4.1, we prioritize the groups with a smaller shareability loss. We select a set of group with the minimum shareability loss for vehicle  $w_x$  (line 14). Then, we put the discarded requests back to working pool  $R_p$  (line 15), where  $w_x.ac$  indicates the current accepted requests of vehicle  $w_x$ . At the end of each acceptance phase, we assign



**Table 3: Experimental Settings.**

Parameters	Values
the number, $n$ , of requests	10K, 50K, 100K, 150K, 200K, <b>250K</b>
the number, $m$ , of vehicles	1K, 2K, 3K, 4K, <b>5K</b>
the capacity of vehicles $c_r$	2, 3, <b>4</b> , 5, 6
the deadline parameter $\gamma$	0.2, 0.3, <b>0.5</b> , 0.8, 1.0
the penalty coefficient $p_r$	2, 5, <b>10</b> , 20, 30
the batching time $\Delta$ (s)	1, 3, <b>5</b> , 7, 9

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  $\text{cost}(r_1)$  and  $\text{cost}(r_1) + \text{cost}(c, a)$ , respectively. The priority sequence in  $r_1$ ’s candidate queue is  $\langle w_2, w_1 \rangle$ . In this way, we can calculate the candidate queue of the remaining requests, as shown in the 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 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 be updated 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  $\frac{\text{cost}(P)}{\sum_{r \in S_G} \text{cost}(r)}$ . Therefore,  $r_2$  will be discarded and propose in the third round, and  $w_2$  will take  $\{r_4, r_2\}$  by Theorem 4.2. Finally,  $w_1$  and  $w_2$  was assigned with  $\{r_1, r_3\}$  and  $\{r_2, r_4\}$ , respectively.

**Complexity Analysis.** With grid index, we can retrieve the candidate vehicles in a constant time for each request. 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 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.

## 5 EXPERIMENTAL STUDY

### 5.1 Data Set

The road networks of *Chengdu* and *NYC* are retrieved as undirected graphs. 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 in [32] with LRU cache [25] for all algorithms. The details of the road networks are shown in Table 4.

**Table 4: Details of Road Networks.**

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

We conduct experimental study on two real-life datasets used in [43]. The first dataset is the requests collected by Didi in Chengdu, China (noted as *Chengdu*). 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. We extracted the requests of the day with the largest number of requests as our request data (November 18, 2016 and April 09, 2016). Since there is no deadline field in the datasets, we generate the deadline of each request by adding an additional buffer time based on the release time, which is proportional to the request’s length with a coefficient  $\lambda$  [45]. For example, we set the deadline of a request  $r_i$  with  $d_i = t_i + (1 + \gamma) * \text{cost}(r_i)$ . In practice, the waiting time accepted by the request is limited. Therefore, we set the maximum waiting time threshold for requests to be 5min, that is,  $w_i = \min(5\text{min}, d_i - \text{cost}(r_i) - t_i)$ . The distribution of the sources and destinations of the datasets are shown in Figure 9. The source and destination of the requests are mapped to the corresponding node of the road network in advance. The initial position of the vehicle is randomly selected from the road network. The number of vehicles changes according to the experimental settings. In addition, we set all vehicles to have the same capacity controlled by parameter  $c$ . In the following experiments, we use Chengdu’s data for a whole day and NYC for half a day (with 250k requests for each dataset). Table 3 shows the parameter settings, and default values are in bold.

### 5.2 Approaches and Measurements

We compare our algorithm with the following algorithms:

- **pruneGDP** [43]. It inserts the request into the vehicle’s current schedule sequentially and selects the vehicle with the least increased distance for service.
- **RTV** [4]. It processes the requests in batch mode and obtains an allocation scheme with the least increased distance and the most number of serving vehicles by linear programming.
- **GAS** [52]. It divides requests into batches and enumerates all combinations for each vehicle’s candidate requests in random order. Then, it selects the most profitable group for each vehicle. The lengths of all requests in the group are used as the profit.
- **RAND**. It randomly arranges requests to a feasible vehicle sequentially, and the trip schedule of the vehicle is maintained by the linear insertion method in [43].

We evaluate the unified cost, service rate and overall response time of all algorithms. Specifically, the unified cost adopts the evaluation of total revenue in [43], and the varying penalty coefficient  $p_r$  is equivalent to the balance between income per unit time and fare per unit distance. 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* [1] to solve the linear programming in *RTV* algorithm and all algorithms are implemented in single thread.

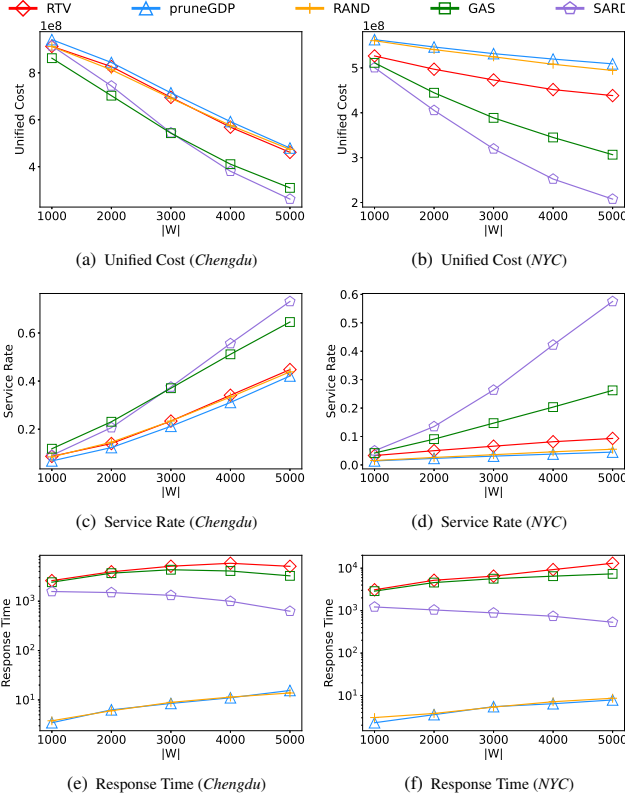


Figure 7: Performance of varying  $|W|$ .

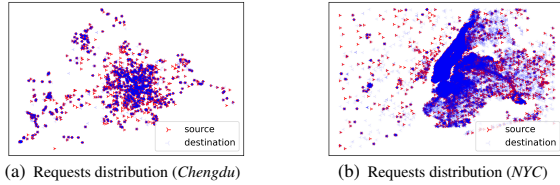


Figure 9: The distribution of datasets.

### 5.3 Experimental Results

**Effect of the number of vehicles.** Figure 7 shows the results of varying the number of vehicles from 1K to 5K. With the increasing number of vehicles, all the results are getting better. In terms of unified cost, SARD and GAS are ahead of other methods, and the gap between these two algorithms and other algorithms is gradually widening. In the *Chengdu* dataset, the results of the two algorithms are very close, but SARD achieves an improvement of 2.09% ~ 59.22% compared with other methods in the *NYC* dataset. As for serve rate, since the penalty part of the unified cost caused by the rejected requests decreasing when the algorithm achieves a higher overall service rate, the gap between the algorithms is consistent with the trend of unified cost. Compared with other existing methods, our method can obtain 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 response 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.

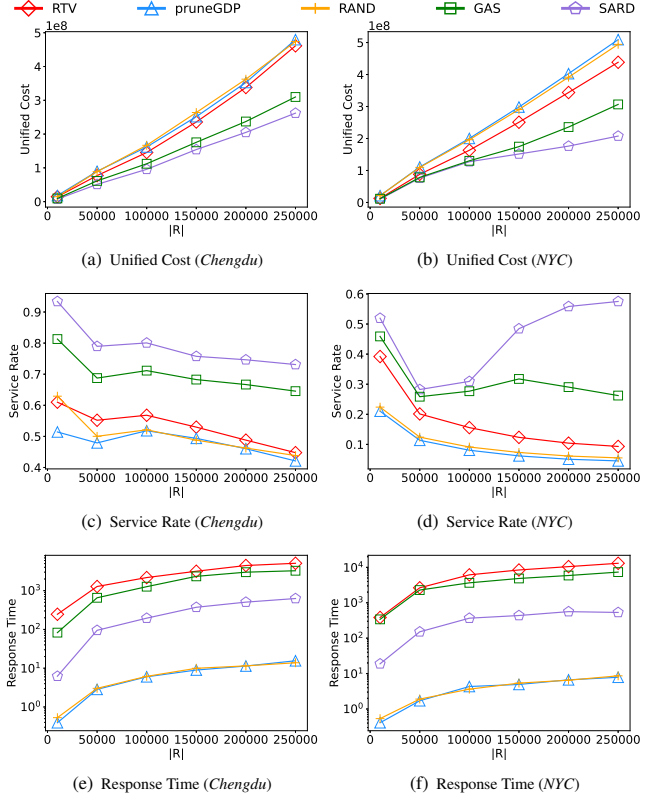


Figure 8: Performance of varying  $|R|$ .

**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% ~ 52.99% at most compared with pruneGDP. Besides, compared with the state-of-art batch-based method GAS, SARD can achieve up to 12.09% and 31.27% higher serve rates on the two datasets, respectively. For response time, the insertion-based methods are still faster. But among the batch-based methods (i.e., SARD, RTV and GAS), SARD is 4.21 $\times$  ~ 38.6 $\times$  faster than GAS and RTV on two datasets.

**Effect of deadline.** Figure 10 presents the results of varying deadline of requests by changing the deadline parameter  $\gamma$  from 0.2 to 1. For service rate, the results of SARD is similar to the existing algorithms when we set the deadline of requests strictly, i.e.,  $\gamma = 0.2$ . The reason is that the candidate vehicles for each request are greatly reduced with a minor deadline, then it is difficult to apply the grouping strategy in batch mode to improve the performance significantly. 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 *Chengdu* 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 inefficient on *NYC* dataset with  $\gamma = 2.0$ , which is mainly caused by

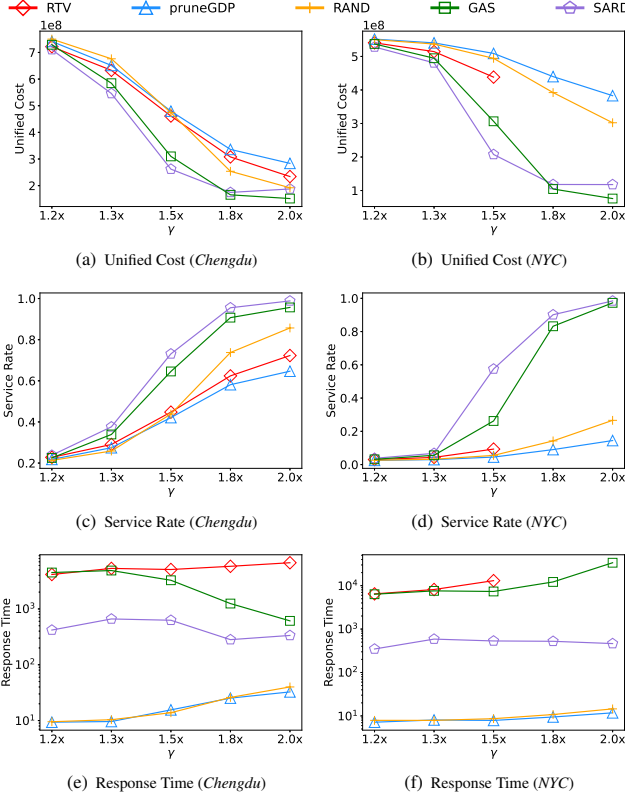


Figure 10: Performance of varying  $\gamma$ .

increasing of the number of request combinations with the relaxation of the deadline. Besides, GAS enumerates all the combinations of requests and schedules almost for each vehicle. The number of candidate requests increases as the number of requests increases. However, in SARD, the requests are proposed to different vehicles in a more decentralized way in each round, and 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 greatly 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* [1]. Meanwhile, SARD performs the best in terms of unified cost, which saves up to 48.03% and 73.16% compared with other existing algorithms on two datasets. For the response time, RAND and pruneGDP are the fastest, while SARD is the runner-up. SARD is  $1.83\times \sim 72.68\times$  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 Chengdu dataset. This is because the increase of vehicle capacity will not affect the process of random selection of vehicles, which has not considered the different increased cost 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 large capacity than RAND. For a similar

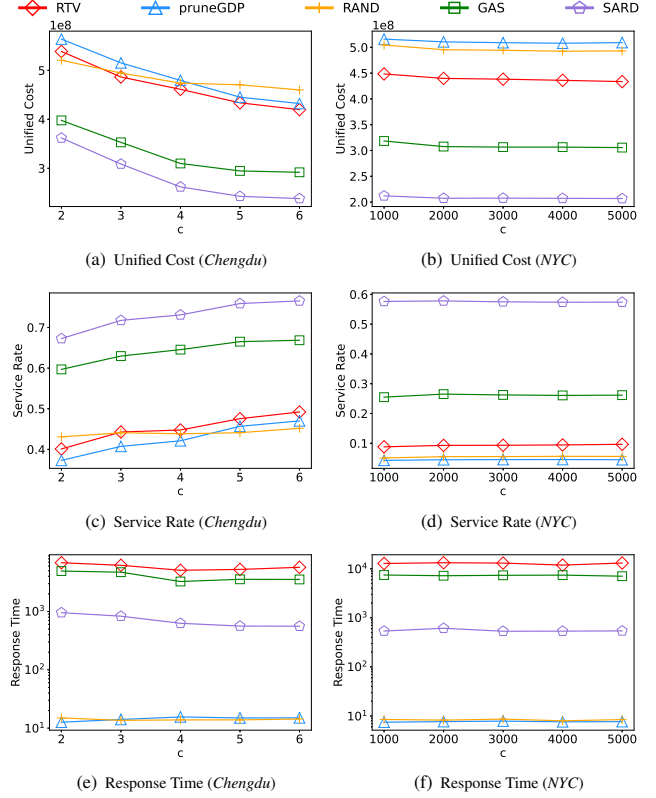


Figure 11: Performance of varying  $c_r$ .

Table 5: Performance of Pruning Strategies.

Method	Dataset	#Shortest Path Queries	Time (s)
SARD	Chengdu	2,971,425,109	699.752
	NYC	1,373,047,917	770.588
SARD-O1	Chengdu	2,839,734,483	662.048
	NYC	1,270,116,743	750.598
SARD-O2	Chengdu	2,687,618,179	626.864
	NYC	1,012,272,255	595.096

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 still is the best among all tested algorithms (7.58% ~ 53.39% higher service rate than other tested algorithms). In terms of response time, SARD still is the fastest in batch-based methods (RTV, GAS and SARD), which is  $5.17\times \sim 17.52\times$  faster than RTV and GAS.

Due to space limitation, please refer to Appendix A of our technical report for the results and discussion about the experiments on varying penalty coefficient  $p_r$  and the batching time  $\Delta$ .

**Effects of pruning strategies.** Table 5 shows the effect of the pruning strategies proposed in Section 3.2 with default parameters. We noted the result without pruning strategies as *SARD*, *SARD-O1* with the Angle pruning, *SARD-O2* with Angle and Ellipse pruning proposed in Section 3.2. The pruning strategies of SARD-O2 save 9.55% and 26.28% of the shortest path queries and save 10.42% and 22.77% of the total running time on the two datasets compared with SARD, respectively. Besides, it has no noticeable impact on the service rate and unified cost.

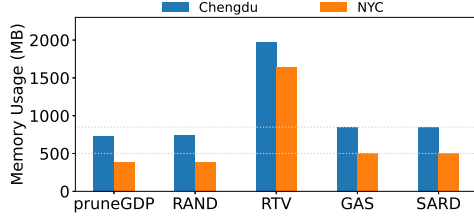


Figure 12: Memory Consumption.

**Memory consumption.** The main occupation of memory comes from the storage of road network, grid index, and cache, so it is constant in most cases. Therefore, we showed the memory usage under the default parameters here (Figure 12). The online mode algorithms follow the first-come-first-serve mode and use the least memory. In contrast, the batch mode algorithms need additional storage to store the batch’s requests’ combinations (e.g., RTV-Graph for RTV, additive index for GAS, shareability graph for SARD). Since RTV rely on an integer linear program, the memory required is more than twice that of GAS and SARD. In addition, the cost of storing an undirected and unweighted shareability graph is cheap, so the memory used in SARD is similar to 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. Different from GAS, our SARD only enumerates combinations among the proposed requests for each vehicle, thus  $n$  is much smaller than it is 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 Chengdu, which means that in the same batch time, the algorithms based on the combination enumeration strategy (SARD and GAS) can obtain more candidate schedules to perform better in NYC. In addition, NYC’s road network is more compact than Chengdu (the nodes are only half of Chengdu), and the requests are more concentrated. Therefore, the requests in NYC have a higher chance of pairing in NYC. Thus, SARD performed better than Chengdu 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 [9]) and apply a tumbling window strategy to extend to the distribution environment.

**Summary:** (1) The batch-based methods can get less unified costs and higher service rates than the online-based methods. For instance, the service rate of SARD is up to 50% higher than that of other tested methods. (2) SARD runs up to 72.68x faster than the

other batch-based methods. For example, in Figure 10(f), SARD can process 12 hours requests in 8 minutes, but GAS takes up to 9 hours.

## 6 RELATED WORK

With the development of GPS equipment, mobile Internet, and sharing economy, ridesharing service has gradually become an indispensable choice for people to travel. The ridesharing problem can be reduced to a variant of the Dial-a-Ride (DARP) problem [18, 47], 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 service can be classified into static or dynamic according to whether all user requests are known in advance. Most of the existing works [17, 48] on DARP are working in static environment. For the dynamic ridesharing problem, the existing solutions are mainly in online mode [24, 36, 43, 49] or batch mode [4, 52, 54, 55]. In online mode, *insertion*[28] is the state-of-the-art operation of the existing works [22, 26] in route planning, which inserts the pickup and drop-off locations of the request into the vehicle’s schedule without reordering. The insertion-based algorithms are more efficient in practice on real-life datasets [49]. Furthermore, in [43], Tong *et al.* proposed an improved insertion method based on dynamic programming, which checks the constraints in constant time and dispatches requests in linear time. On the other hand, Huang *et al.* proposed the structure of kinetic tree in [24], which is used to trace all feasible routes for each vehicle to reduce the total drive distance. With the kinetic tree, they always provide the optimal schedule for vehicles whenever the schedule changes (i.e., new rider arrives). For batch-based algorithms, they usually partition the request into groups with a schedule, and then assign groups to their appropriate vehicles. In [4], Alonso-Mora *et al.* propose RTV-Graph to model the relationship and constraints among requests, trips and vehicles, where trips are the groups composed of shareable requests. With the RTV-Graph, they minimize the utility function by linear programming to get the allocation result between vehicles and trips. But the time cost for enumerating trips in the process of building RTV-Graph grows exponentially. In [52], Zeng *et al.* proposed an index called additive tree for pruning the infeasible groups during the group enumeration, and greedily choose the most profitable request group to server for each vehicle. However, no existing work takes the priority of serving riders in the same batch into account. We proposed the structure of shareability graph and propose a two-phase heuristic algorithm, namely SARD, which exploits the structures to prioritize the requests in the assignment.

## 7 CONCLUSION

In this paper, we study the dynamic ridesharing problem with 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 better service rate, less unified cost with less running time compared with the state-of-the-art batch-based methods [4, 52].



## REFERENCES

- [1] 2016. [Online] GNU GLPK . <https://www.gnu.org/software/glpk/>.
- [2] 2018. [Online] Didi Chuxing . <https://www.didiglobal.com/>.
- [3] 2018. [Online] uberPOOL . <https://www.uber.com/>.
- [4] Javier Alonso-Mora, Samitha Samaranyake, Alex Wallar, Emilio Frazzoli, and Daniela Rus. 2017. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proc. Natl. Acad. Sci. USA* 114, 3 (2017), 462–467. <https://doi.org/10.1073/pnas.1611675114>
- [5] Mohammad Asghari, Dingxiong Deng, Cyrus Shahabi, Ugur Demiryurek, and Yaguang Li. 2016. 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, Burlingame, California, USA, October 31 - November 3, 2016*. 3:1–3:10.
- [6] Mohammad Asghari and Cyrus Shahabi. 2017. 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*. 7:1–7:10.
- [7] Xiaohui Bei and Shengyu Zhang. 2018. Algorithms for Trip-Vehicle Assignment in Ride-Sharing. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 3–9. <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16583>
- [8] Jayaram Bhasker and Tariq Samad. 1991. The clique-partitioning problem. *Computers & Mathematics with Applications* 22, 6 (1991), 1–11.
- [9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. <http://sites.computer.org/debull/A15dec/p28.pdf>
- [10] Lu Chen, Chengfei Liu, Kewen Liao, Jianxin Li, and Rui Zhou. 2019. Contextual Community Search Over Large Social Networks. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 88–99. <https://doi.org/10.1109/ICDE.2019.00017>
- [11] Lu Chen, Chengfei Liu, Rui Zhou, Jianxin Li, Xiaochun Yang, and Bin Wang. 2018. Maximum Co-located Community Search in Large Scale Social Networks. *Proc. VLDB Endow.* 11, 10 (2018), 1233–1246. <https://doi.org/10.14778/3231751.3231755>
- [12] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. 2010. Finding maximal cliques in massive networks by H\*-graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 447–458. <https://doi.org/10.1145/1807167.1807217>
- [13] Peng Cheng, Hao Xin, and Lei Chen. 2017. Utility-Aware Ridesharing on Road Networks. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1197–1210. <https://doi.org/10.1145/3035918.3064008>
- [14] Peng Cheng, Hao Xin, and Lei Chen. 2017. Utility-aware ridesharing on road networks. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1197–1210.
- [15] Deming Chu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang, Yinglong Xia, and Chenyi Zhang. 2020. Finding the Best k in Core Decomposition: A Time and Space Optimal Solution. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 685–696. <https://doi.org/10.1109/ICDE48307.2020.00065>
- [16] Blerim Cici, Athina Markopoulou, and Nikolaos Laoutaris. 2015. Designing an on-line ride-sharing system. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*. 60:1–60:4.
- [17] Jean-François Cordeau. 2006. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research* 54, 3 (2006), 573–586.
- [18] Jean-François Cordeau and Gilbert Laporte. 2003. The dial-a-ride problem (DARP): Variants, modeling issues and algorithms. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies* 1, 2 (2003), 89–101.
- [19] Jean-François Cordeau and Gilbert Laporte. 2003. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological* 37, 6 (2003), 579–594.
- [20] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis (Eds.). ACM, 589–598. <https://doi.org/10.1145/3178876.3186125>
- [21] Christos Giatsidis, Fragkiskos D. Malliaros, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2014. CoreCluster: A Degeneracy Based Graph Clustering Framework. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, Carla E. Brodley and Peter Stone (Eds.). AAAI Press, 44–50. <http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8530>
- [22] Lauri Häme. 2011. An adaptive insertion algorithm for the single-vehicle dial-a-ride problem with narrow time windows. *European Journal of Operational Research* 209, 1 (2011), 11–22.
- [23] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 1311–1322. <https://doi.org/10.1145/2588555.2610495>
- [24] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. 2014. Large Scale Real-time Ridesharing with Service Guarantee on Road Networks. *PVLDB* 7, 14 (2014), 2017–2028. <https://doi.org/10.14778/2733085.2733106>
- [25] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. 2014. Large Scale Real-time Ridesharing with Service Guarantee on Road Networks. *Proc. VLDB Endow.* 7, 14 (2014), 2017–2028. <https://doi.org/10.14778/2733085.2733106>
- [26] Irina Ioachim, Jacques Desrosiers, Yvan Dumas, Marius M Solomon, and Daniel Villeneuve. 1995. A request clustering algorithm for door-to-door handicapped transportation. *Transportation science* 29, 1 (1995), 63–78.
- [27] Svante Janson, Tomasz Łuczak, and Ilkka Norros. 2010. Large cliques in a power-law random graph. *Journal of Applied Probability* 47, 4 (2010), 1124–1135.
- [28] Jang-Jei Jaw, Amedeo R Odoni, Harilaos N Psarafitis, and Nigel HM Wilson. 1986. A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. *Transportation Research Part B: Methodological* 20, 3 (1986), 243–257.
- [29] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA (The IBM Research Symposia Series)*, Raymond E. Miller and James W. Thatcher (Eds.). Plenum Press, New York, 85–103. [https://doi.org/10.1007/978-1-4684-2001-2\\_9](https://doi.org/10.1007/978-1-4684-2001-2_9)
- [30] Maksim Kitsak, Lazaros K Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H Eugene Stanley, and Hernán A Makse. 2010. Identification of influential spreaders in complex networks. *Nature physics* 6, 11 (2010), 888–893.
- [31] Jianxin Li, Xinxue Wang, Ke Deng, Xiaochun Yang, Timos Sellis, and Jeffrey Xu Yu. 2017. Most Influential Community Search over Large Social Networks. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 871–882. <https://doi.org/10.1109/ICDE.2017.136>
- [32] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An Experimental Study on Hub Labeling based Shortest Path Algorithms. *Proc. VLDB Endow.* 11, 4 (2017), 445–457. <https://doi.org/10.1145/3186728.3164141>
- [33] Zijian Li, Lei Chen, and Yue Wang. 2019. G\*-Tree: An Efficient Spatial Index on Road Networks. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 268–279. <https://doi.org/10.1109/ICDE.2019.00032>
- [34] Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. 2021. Hierarchical Core Maintenance on Large Dynamic Graphs. *Proc. VLDB Endow.* 14, 5 (2021), 757–770. <http://www.vldb.org/pvldb/vol14/p757-zhang.pdf>
- [35] Qingyuan Linghu, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2020. 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 [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2211–2226. <https://doi.org/10.1145/3318464.3389744>
- [36] Shuo Ma, Yu Zheng, and Ouri Wolfson. 2013. T-share: A large-scale dynamic taxi ridesharing service. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 410–421. <https://doi.org/10.1109/ICDE.2013.6544843>
- [37] Shuo Ma, Yu Zheng, and Ouri Wolfson. 2015. Real-Time City-Scale Taxi Ridesharing. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1782–1795. <https://doi.org/10.1109/TKDE.2014.2334313>
- [38] James Pan, Guoliang Li, and Juntao Hu. 2019. Ridesharing: Simulator, Benchmark, and Evaluation. *Proc. VLDB Endow.* 12, 10 (2019), 1085–1098. <https://doi.org/10.14778/3339490.3339493>
- [39] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. 2012. Community detection in Social Media - Performance and application considerations. *Data Min. Knowl. Discov.* 24, 3 (2012), 515–554. <https://doi.org/10.1007/s10618-011-0224-z>
- [40] Oleg Rokhlenko, Ydo Wexler, and Zohar Yakhini. 2007. Similarities and differences of gene expression in yeast stress conditions. *Bioinform.* 23, 2 (2007),



- 184–190. <https://doi.org/10.1093/bioinformatics/btl308>
- [41] Douglas Oliveira Santos and Eduardo Candido Xavier. 2013. 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*, Francesca Rossi (Ed.). IJCAI/AAAI, 2885–2891. <http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6779>
  - [42] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. Corescope: Graph mining using k-core analysis—patterns, anomalies and algorithms. In *2016 IEEE 16th international conference on data mining (ICDM)*. IEEE, 469–478.
  - [43] Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, Jieping Ye, and Ke Xu. 2018. A Unified Approach to Route Planning for Shared Mobility. *PVLDB* 11, 11 (2018), 1633–1646. <https://doi.org/10.14778/3236187.3236211>
  - [44] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proc. VLDB Endow.* 5, 9 (2012), 812–823. <https://doi.org/10.14778/2311906.2311909>
  - [45] Jiachuan Wang, Peng Cheng, Libin Zheng, Chao Feng, Lei Chen, Xuemin Lin, and Zheng Wang. 2020. Demand-Aware Route Planning for Shared Mobility Services. *Proc. VLDB Endow.* 13, 7 (2020), 979–991. <https://doi.org/10.14778/3384345.3384348>
  - [46] Kai Wang, Xin Cao, Xuemin Lin, Wenjie Zhang, and Lu Qin. 2018. Efficient Computing of Radius-Bounded k-Cores. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 233–244. <https://doi.org/10.1109/ICDE.2018.00030>
  - [47] Nigel HM Wilson, RW Weissberg, BT Higonnet, and J Hauser. 1975. *Advanced dial-a-ride algorithms*. Technical Report.
  - [48] Ka-Lo Wong and Michael GH Bell. 2006. Solution of the Dial-a-Ride Problem with multi-dimensional capacity constraints. *International Transactions in Operational Research* 13, 3 (2006), 195–208.
  - [49] Yi Xu, Yongxin Tong, Yexuan Shi, Qian Tao, Ke Xu, and Wei Li. 2019. An Efficient Insertion Operator in Dynamic Ridesharing Services. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 1022–1033. <https://doi.org/10.1109/ICDE.2019.00095>
  - [50] San Yeung, Evan Miller, and Sanjay Madria. 2016. A Flexible Real-Time Ridesharing System Considering Current Road Conditions. In *IEEE 17th International Conference on Mobile Data Management, MDM 2016, Porto, Portugal, June 13-16, 2016*. 186–191.
  - [51] Minji Yoon, Bryan Hooi, Kijung Shin, and Christos Faloutsos. 2019. 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, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 647–657. <https://doi.org/10.1145/3292500.3330946>
  - [52] Yuxiang Zeng, Yongxin Tong, Yuguang Song, and Lei Chen. 2020. The Simpler The Better: An Indexing Approach for Shared-Route Planning Queries. *Proc. VLDB Endow.* 13, 13 (2020), 3517–3530. <http://www.vldb.org/pvldb/vol13/p3517-zeng.pdf>
  - [53] Chen Zhang, Ying Zhang, Wenjie Zhang, Lu Qin, and Jianye Yang. 2019. Efficient Maximal Spatial Clique Enumeration. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 878–889. <https://doi.org/10.1109/ICDE.2019.00083>
  - [54] Libin Zheng, Lei Chen, and Jieping Ye. 2018. Order dispatch in price-aware ridesharing. *Proceedings of the VLDB Endowment* 11, 8 (2018), 853–865.
  - [55] Libin Zheng, Peng Cheng, and Lei Chen. 2019. Auction-based order dispatch and pricing in ridesharing. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1034–1045.

## 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 takes 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), it will have a certain effect on the result of LP when the penalty coefficient is small, but the varying of 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% ~ 52.99%. Similarly, since the penalty coefficient does not affect the assignment phase, the execution time does not change obviously on two datasets.

**Effect of batching period.** Figure 14 represents the results of varying the batching period from 1 to 9 seconds for batch-based methods. The varying of batch time has little effect on the batch-based algorithms in terms of unified cost and service rate. SARD performs the best with the varying of batching period, which saves up to 44.18% and 53.08% in unified cost and increases the service rate with up to 29.2% and 48.96% in two datasets, respectively. For running time, since the number of execution rounds will decreases when the time 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.

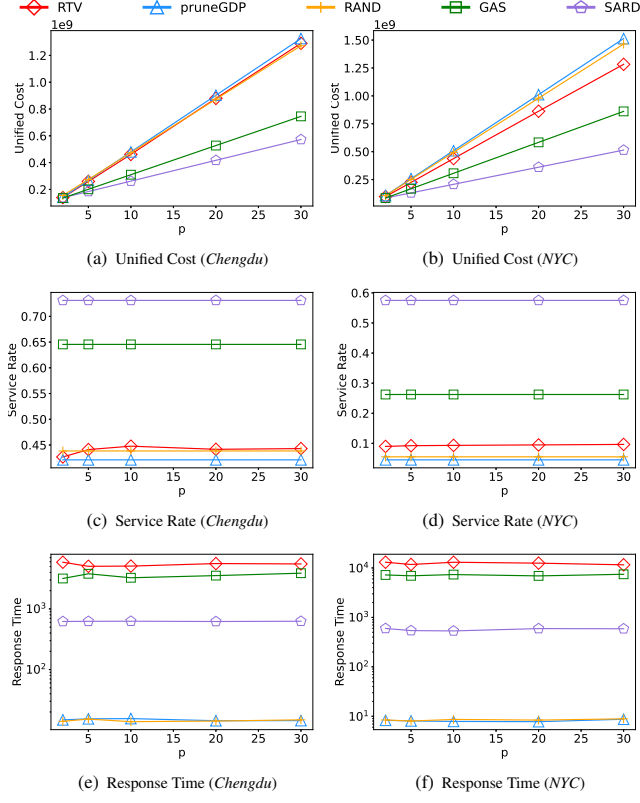


Figure 13: Performance of varying  $p_r$ .

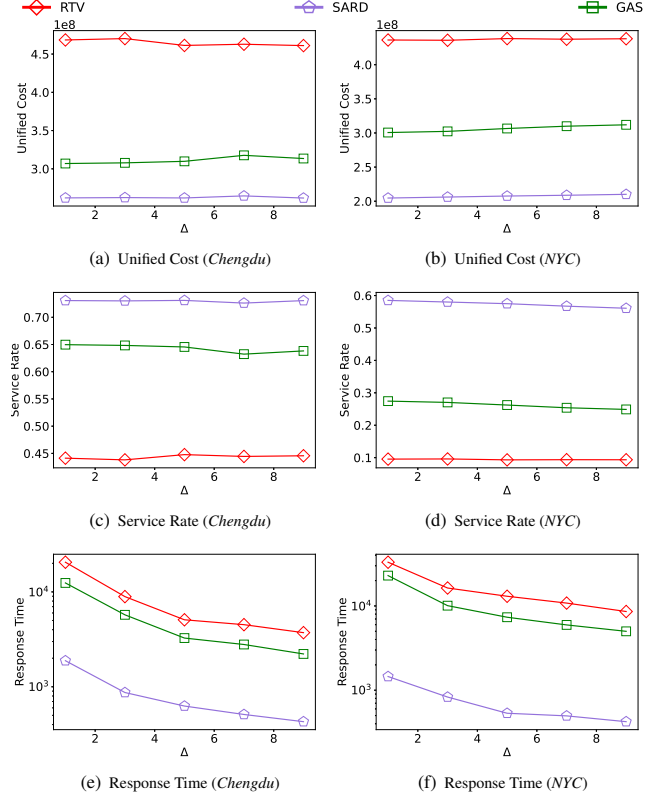


Figure 14: Performance of varying  $\Delta$ .