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

Jiexi Zhan *, Yu Chen *, Peng Cheng *, Lei Chen ◇ †, Wangze Ni †, Xuemin Lin ‡
*East China Normal University, Shanghai, China
◇HKUST (GZ), Guangzhou, China
†HKUST, Hong Kong SAR, China
‡Shanghai Jiaotong University, Shanghai, China
{jxzhan, yu.chen}@stu.ecnu.edu.cn; pcheng@sei.ecnu.edu.cn; leichen@cse.ust.hk; wangzeni@ust.hk; xuemin.lin@gmail.com

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

***Index Terms*—component, formatting, style, styling, insert.**

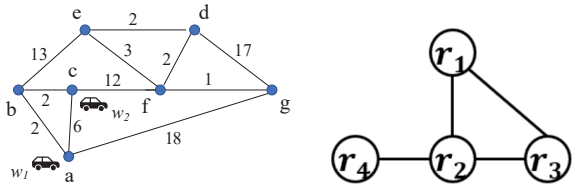
I. Introduction

Recently, ridesharing has become a popular public transportation choice, which greatly reduces energy and relieves traffic pressure. In ridesharing, a driver can serve different riders simultaneously once riders can share parts of their trips and there are enough seats for them. The ridesharing service providers (e.g., Uber [1], Didi [2]) are constantly pursuing better service quality, such as higher service rate [3], [4], [5], lower total travel distance [6], [7], [5], or higher total revenue [8], [9]. For ridesharing platforms, *vehicle-request matching* and *route planning* are two critical issues to address. With a set of vehicles and requests, vehicle-request matching filters out a set of valid candidate requests for each vehicle, while route planning targets on designing a route schedule for a vehicle to pick up and drop off the assigned requests. If a vehicle can serve two requests on the same trip, we call that they have a shareable relationship.

The structure information of graphs reveal key properties (e.g., k-core [10], [11], k-truss [12], [13], clique [14], [15]), aiding in specific applications (e.g., community discovery [16], [17], [18], anomaly detection [19], [20] and influential analysis [21], [22]). In ridesharing problems, the shareable relationships form a *shareability graph* [23], [24], [25], [26], [27], [28], where each node represents a request and each edge indicates that the connected requests can share part of their trips. However, to the best of our knowledge, no existing works have targeted improving service quality by utilizing the *structural properties* of the shareability graph in ridesharing problems.

TABLE I: Requests Release Detail.

request	source	destination	release time	deadline
r_1	a	d	0	30
r_2	c	f	1	19
r_3	b	e	2	21
r_4	c	g	3	21



(a) Road Network (b) Shareability Graph

Fig. 1: A Motivation Example

In real platforms, vehicles and requests often arrive dynamically. Existing works on ridesharing can be summarized into two modes: the *online mode* [29], [30], [31], [6], [7], [32] and *batch mode* [27], [33], [34], [35]. The state-of-the-art operator for the online mode is *insertion* [32], [31], [36], [34], [37], which inserts the requests' sources and destinations into proper positions of the vehicle's route without reordering its current schedule, minimizing the increase in the total travel cost. Batch-based methods [27], [33], [38], [39], [34] first package the incoming requests into groups, then assign a vehicle with a properly designed serving-schedule for each group of requests.

We illustrate our motivation with Example 1. The existing approaches have their shortcomings. Insertion operator is fast but only achieve the local optimal schedule for a request. The batch-based methods can achieve better results than online methods for a period of a relatively long time (e.g., one day), but their time complexities are much higher than insertion (e.g., RTV [27] requires costly bipartite matching). Can we

have a more efficient batch-based approach to achieve better results for a relatively long period? In this paper, we exploit the graph structures in the shareability graph of requests to achieve better results for ridesharing services.

Example 1. *Suppose there are two vehicles, w_1 and w_2, and four requests, r_1 to r_4, on a ridesharing platform. The sources, destinations, release times, and deadlines of the requests are listed in Table I. As shown in Figure 1(a), the vehicles are initially located at points a and c on the road network with seven vertices, each with a capacity of three. In the online-based solution [29], [30], [31], [6], [7], [32], r_2 will be inserted into the existing schedule of r_1 of w_1 with a scheduled route $\langle a, c, f, d\rangle$ at time 1. This results in r_4 not being served as it cannot be inserted into the planned route $\langle c, b, e\rangle$ of w_2, which serves r_3. Although it can provide better scheduling, the existing batch-based methods [27], [38], [39], [34], [5] list all possible request combinations or vehicle-request pair, and run the time-consuming global matching.*

Instead of considering all request combinations, we can heuristically achieve good dispatch result by investigating the shareability graph shown in Figure 1(b). We construct the graph by connecting the requests that can share a vehicle, so the request with a larger degree has more sharing opportunities. If we give higher priority to grouping the requests with lower degree, r_4 will first share w_2 with r_2, then r_3 will share w_1 with r_1. All the requests can be served with scheduled routes $\langle a, b, e, d\rangle$ and $\langle c, f, g\rangle$ for w_1 and w_2, respectively.

In this paper, we propose StructRide, a well-tailored batch-based framework to exploit the structure information of requests' shareability graph in ridesharing systems. The achieved results are significantly improved (e.g., higher service rates and lower travel distances) under just a slight delay of response time. *To the best of our knowledge, we are the first to integrate traditional allocation algorithms with graph analyses of the shareability graph in ridesharing problems, which is also the key insight of our StructRide framework.* In StructRide, we first propose a fast shareability graph builder that efficiently extracts shareable relationships between requests through spatial indexes and an angle-based pruning strategy to only maintain the feasible shareable relationships. To exploit the structure information of the shareability graph, we devise the *structure-aware ridesharing dispatch* (SARD) algorithm to take the cohesiveness of requests in the shareability graph (evaluated through the graph structures) into consideration and revise request priorities through analyzing its shareability. SARD's two-phase "proposal-acceptance" strategy avoids invalid group enumerations for any specific vehicle. For vehicle scheduling, we adjust the insertion order of riders' requests based on shareability, maintaining an optimal schedule with linear insertion complexity. Thus, the batch-based method can be more efficient in large-scale ridesharing problems. To summarize, we make the following contributions:

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

II. Problem Definition

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

A. Definitions

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

To guarantee the quality of ridesharing services, the existing studies [40], [31], [34] commonly use the detour ratio, ϵ, to avoid reaching the destination of each ride too late. In particular, riders join ridesharing services to benefit from a discount on the travel fee, but they need to tolerate some detours. In this paper, the delivery deadline d_i of request r_i is calculated by adding a detour tolerance to the shortest travel time (i.e. $d_i = t_i + \gamma * cost(s_i, e_i)$, $\gamma > 1$).

Definition 2 (Schedule)**.** Given a vehicle w_j with m allocated requests $R_j = \{r_1, \ldots, r_m\}$. Let $S_j = \langle o_1, \ldots, o_{2m}\rangle$ define a schedule for w_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 the schedule a *way-point*. A route S_j is *feasible* if and only if it meets these four constraints:

- **Coverage Constraint**. For any request $r_i \in R_j$, the source s_i and destination e_i should be included in S_j.
- **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**. The total number of assigned riders must not exceed the maximum capacity c_j of vehicle v_j.
- **Deadline Constraint**. For any $o_x \in S_j$, the total driving time before o_x must satisfy the inequality 1.

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

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

Definition 3 (Buffer Time [31], [37])**.** Given a valid vehicle schedule $S_i = \langle o_1, o_2, \ldots, o_{2m}\rangle$, let $buf(o_x)$ be the maximum detour time at the way-point o_x without violating the deadline constraints of its consequent way-points.

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

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

Definition 4 (Batched Dynamic Ridesharing Problem (BDRP) [37])**.** Given a set R of n dynamically arriving requests and a set W of m vehicles, the dynamic ridesharing problem is to plan a feasible schedule for each vehicle $w_i \in W$ to minimize a specific utility function in batch mode.

In BDRP, incoming requests in time period T are handled as a batch $\mathcal{P}$. Requests are partitioned into groups $\mathbb{G} = \{G_1, G_2, \ldots, G_z\}$, satisfying following conditions: $\forall G_x, G_y \in \mathbb{G}, G_x \cap G_y = \emptyset$ and $\mathcal{P} = \bigcup_{a=1..z} G_a$. We then assign request groups to valid vehicles to minimize the utility function. In this work, we refer to the unified cost function UC in [37] and define the utility function U:

$$U(W, \mathcal{P}) = \alpha \sum_{w_i \in W} \mu(w_i, G_{w_i}) + \sum_{G_i \in G^-} p_i \tag{3}$$

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

where S_{w_i} is the planned schedule for vehicle w_i and $\mu(w_i, G_{w_i})$ depicts the total travel cost of all schedules. G^- is composed of unassigned request groups in each batch, with a penalty p_i.

By setting special α and p_i, the utility function U can support common optimization objectives in ridesharing problems, such as minimizing total travel cost, maximizing service rate, and maximizing total revenue [37]. In this paper, we fix α to 1 and define $p_i = p_r \sum_{r \in G_i} cost(r.s, r.e)$ to indicate the profit loss caused by unassigned requests with a parameter p_r.

Discussion of Hardness. The existing works [37], [31], [41], [7] have proved that dynamic ridesharing problem is NP-hard, thus intractable. Moreover, It is proved that there is no polynomial-time algorithm with a constant competitive ratio for dynamic ridesharing problem [37]. Thus, we use experimental results to show the effectiveness of our approaches.

We summarize the key notations in Appendix A of our technical report [42].

B. An Overview of StructRide Framework

To efficiently and effectively solve BDRP, we propose a batch-based framework, namely StructRide, that exploits the structure information of the shareability graph of requests to improve the performance of service rates and unified costs. We first introduce the main parts of our StructRide framework, as shown in Figure 2.

Index Structure. In ridesharing, since vehicles keep moving over time, the used index must update efficiently. The grid index allows querying and updating moved vehicles in constant time. In StructRide, 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 radius r in constant time.

Schedule Maintenance. StructRide arranges new request pickup and drop-off locations into available vehicle schedules. Two wildly solutions are used: *kinetic tree insertion* [7] and *linear insertion* [37], [36]. Kinetic tree insertion maintains all

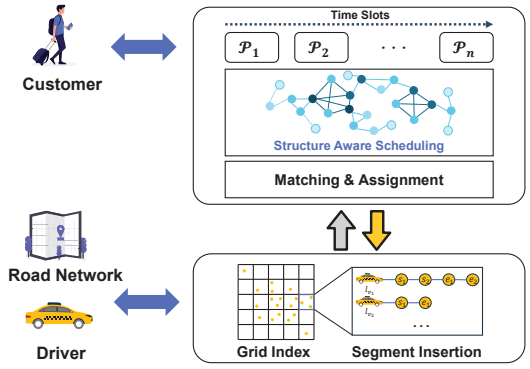


Fig. 2: Overview of System Framework *StructRide*.

feasible schedules for a vehicle in a tree structure, ensuring optimality. Linear insertion maintains only the current optimal schedule, inserting new points without *reordering*. While this may miss the optimal schedule, Ma *et al.* [43] found minimal impact on travel cost reduction, with increased time and space usage for reordering. Thus, we use linear insertion for scheduling.

Structure-Aware Assignment. We process the incoming requests into batches by their release timestamps. For each batch, we utilize the shareability graph in Section III to analyze the sharing relationship between requests. In the *Matching and Assignment* phase, we propose a two-phase algorithm SARD in Section IV. Requests will propose to candidate vehicles in descending order of additional travel cost (i.e., propose to vehicles needing more additional travel costs first). Vehicles then select structure-friendly request groups to accept, using our novel measurement, namely *shareability loss*, to evaluate how merging a group of requests into a supernode affect the sharing probability of the rest nodes in the shareability graph.

III. Shareability Graph Construction

Although the requests in the same batch have similar release times, the sharing probabilities can vary dramatically. A request with higher shareability has more opportunities to share vehicles with other requests. Thus, an intuitive motivation for optimizing the grouping process is to give higher priority to a request with lower shareability. Existing works [23], [24], [25], [26], [27], [28] utilize an effective structure, the *shareability graph*, to model and manage the sharing relationships between requests. In this section, we propose an efficient construction algorithm for it.

A. Observations on Shareability Graph

Definition 5 (Shareability Graph [23], [24], [25], [26], [27], [28])**.** Given a set of m requests R, let $SG = \langle R, E \rangle$ represent a *shareability graph* with nodes R and edges E, where each node in R corresponds to a request. An edge $(r_a, r_b) \in E$ signifies that r_a and r_b are shareable, meaning that there is at least one *feasible* schedule for them to be served by a vehicle.

For instance, in the *shareability graph* depicted in Figure 1(b), the edge $e = (r_1, r_2)$ between r_1 and r_2 indicates that there is a feasible route to serve them in a trip (e.g., a schedule of $\langle s_1, s_2, e_2, e_1 \rangle$). Therefore, we can easily identify

the candidate shareable requests for each request by checking the neighbors in the shareability graph. Additionally, we have the following two observations:

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

Observation 2. *The subgraph induced by the corresponding nodes of potential shareable requests is full connected.* In the shareability graph, if there exists a valid sharing schedule consisting of k requests, there must exist a fully connected subgraph, also called *k-clique* [44], [45], among the corresponding nodes. The reason is that, for a valid sharing schedule, any its two requests must can share with each other, thus a connecting edge between the corresponding nodes will exist in the shareability graph. With this observation, we can prune some infeasible combinations in the grouping phase of each batch.

The shareability graph's structure is intuitive, and some similar structures have been implemented in other works. These designs focus on different approaches to the vehicle-request matching problem. Wang et al. [46] developed a tree cover problem to minimize vehicle use for urban demands. Alonso-Mora et al. [27] utilize linear programming to optimally assign vehicles to shareable customer groups. Zhang et al. [47] approach passenger matching as a monopartite matching problem, solved by the Irving-Tan algorithm. *However, existing studies ignored the heuristic insights brought by the local structure of the shareability graph to guide the vehicle-request matching.* Furthermore, they construct the shareability graphs through the brute force enumeration of requests pairs without careful tailoring. Therefore, we propose an efficient shareability graph construction method.

B. Angle Pruning Strategy

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 for a valid sharing schedule using a linear insertion method. The efficiency of this construction algorithm is primarily dominated by the number of shortest path queries needed. To avoid enumerating all pairs of requests, we suggest an angle pruning strategy, illustrated in Figure 3, for more efficient shareability graph construction. Note that, when we prune the candidate requests for a given request r_a, to avoid duplicated consideration of schedules, we only consider the schedules with s_a (the source of r_a) as the first way-point.

Our angle pruning strategy is based on the travel directions of the requests. We notice that the requests with similar travel directions are more likely to share trips. For instance, a northward request is unlikely to share a trip with a southward request because the driver would need to turn around and spend considerable time driving in the opposite direction. We define a threshold δ to prune request pairs that have similar

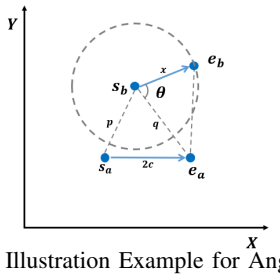


Fig. 3: An Illustration Example for Angle Pruning

sources but divergent directions with Theorem III.1. Let $\overrightarrow{ab}$ be the vector pointing from a to b.

Theorem III.1. *For a request r_a and its candidate sharing request r_b, the expected probability of sharing a trip will increase when the angle θ between $\overrightarrow{s_b e_a}$ and $\overrightarrow{s_b e_b}$ decreases.*

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

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

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

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

For instance, we fit the probability density function of the requests' distances of Chengdu and NYC datasets with a log-normal distribution. The probability expectations $\mathbb{E}(\theta \geq \frac{\pi}{2})$ are 40.98% and 41.38% respectively when $\gamma = 1.5$.

Discussion. Our angle pruning strategy is an approximate pruning method, meaning it might occasionally remove some feasible shareable pairs. However, our experiments shows that this false-negative pruning will not harm the final results.

Algorithm 1: Dynamic *Shareability Graph* Builder

Input: The previous sharability graph SG' and its request set R_p, a new request batch $\mathcal{P}$, an angle threshold δ
Output: The corresponding shareability graph SG

1 $SG = SG'$
2 **for** $r_a \in \mathcal{P}$ **do**
3 $\quad V = V \cup r_a, R_p = R_p \cup r_a$
4 $\quad C \leftarrow$ candidate requests filtered by spatial indexes, deadline and detour tolerance constraint from R_p
5 $\quad$ **for** $r_b \in C$ **do**
6 $\quad\quad$ **if** $\arccos\left(\frac{\overrightarrow{s_b e_a} \cdot \overrightarrow{s_b e_b}}{|\overrightarrow{s_b e_a}||\overrightarrow{s_b e_b}|}\right) \in \left[-\frac{\delta}{2}, \frac{\delta}{2}\right]$ **then**
7 $\quad\quad\quad$ **if** *r_a and r_b are shareable* **then**
8 $\quad\quad\quad\quad E = E \cup (r_a, r_b)$
9 **return** $SG = \langle V, E \rangle, R_p$

In fact, it can sometimes even improve the final results, such as by achieving higher serving rates. We think this is because the angle pruning strategy can help avoid considering some feasible but not very appropriate shareable pairs (i.e., the directions of the two requests diverge too much) in the following grouping and dispatching stage. This can somehow give the algorithm for the following stage a higher probability of achieving a better final result.

C. Dynamic Shareability Graph Builder

The details of the dynamic shareability graph builder algorithm are illustrated in Algorithm 1.

Firstly, we use the previous batch's sharability graph to initialize the current one (line 1). We try to find new shareable relationships brought by each newly arrived request successively (lines 2-8). For each new request r_a, we first add it to the graph nodes and the current request set (line 3). By leveraging spatial indexes (e.g., grid index), deadline constraints, and detour tolerance, we can quickly obtain a candidate request set C with a similar source location to r_a without a shortest path query. Then, we further filter the requests $r_b \in C$ by the angle pruning rule (lines 5-8). We construct the vector $\overrightarrow{s, e}$ by the source and destination of the request to represent the distance and direction. If the angle θ of $\overrightarrow{s_b e_a}$ and $\overrightarrow{s_b e_b}$ exceeds the given angle threshold δ, r_b will be pruned (line 6). We'll add an edge (r_a, r_b) if r_a and r_b are shareable (lines 7-8). Finally, we get a shareability graph SG with V and E generated in the above steps (line 8).

Complexity Analysis. Assume the shortest path query takes $O(q)$ time. For a request r_a, filtering can be done in constant time by searching *Minimum Bounding Rectangle* of the detour tolerated area in grid index. In the worst case, the candidate request set size $|C| = |R_p| - 1$ in line 3. Besides, the angle calculation takes constant time, requiring only two insertions to test if r_a and r_b are shareable in $O(k)$. Thus, the algorithm's final complexity is $O(q \cdot |R_p| \cdot |\mathcal{P}|)$.

Discussion. The proposed angle pruning strategy is derived based on Euclidean space. In a realistic road network, facilities such as expressways make some excluded solutions actually feasible. However, our experiments show very few cases are discarded, making the angle pruning strategy acceptable for heuristic pruning of the shareability network.

IV. Structure-Aware Dispatching

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

A. Schedule Maintenance

There are two state-of-art strategies for schedule maintenance in previous work: kinetic tree insertion [7] and linear insertion [37], [36]. The kinetic tree maintains all feasible schedules and checks all available way-points ordering exhaustively to insert a new request. While the kinetic tree can achieve 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 to the vehicle). In contrast, the schedule obtained by the linear insertion method is optimal only for the current schedule (i.e., local optimal). Linear insertion is optimal when the number of requests is 2. In the experimental study in Section V, we find that in NYC and Chengdu datasets, if we use linear insertion to handle requests according to their release time, the probability of achieving a global optimal schedule is up to 89% and 85%, when inserting the third and fourth request, respectively. To improve such a probability with the linear insertion method, we reorder the insertion sequence of the requests based on Observation 1 in Section III-A. Specifically, we first select two requests with the lowest shareability (i.e., the degree of the node) in the shareability graph and generate an optimal sub-schedule. Then, we insert the remaining requests into the sub-schedule one by one in ascending order of their shareability. In this way, we improve the probability of achieving an optimal schedule using the linear insertion method to 91% and 90%.

B. Request Grouping

In batch mode, we enumerate all feasible request combinations before the assignment phase. Listing all combinations requires checking $\sum_{i=1}^{c} \binom{n}{i}$ groups, where c is vehicle capacity. After that, we verify the existence of a feasible route to serve these requests by linear insertion [37] in $O(k^2)$ per group, where k is the group size. However, many groups are invalid for vehicles in practice. To avoid unnecessary enumeration, Zeng *et al.* proposed an index called *additive tree* [33], which enumerates valid groups level by level through a tree-based structure. In the additive tree, each node represents a valid group of requests, extending its parent node's group by adding one more request.

Although the additive tree helps prune some invalid groups in the enumeration, it still maintains all feasible schedules for each tree node. However, we will only pick the best one of these schedules for each group at the end. With the schedule maintenance method proposed in Section IV-A, we

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

can heuristically reorder the insertion sequence to get an optimal schedule with higher probability. *Note that, we still do not alter existing schedules, and just reorder the sequence to insert the new requests to the existing schedules one by one.* In building the modified additive tree, we keep only one schedule for each node by inserting a new request to its parent's schedule.

Algorithm 2 outlines the detailed construction steps. Initially, c empty sets $\{RG_1, RG_2, \cdots, RG_c\}$ are initialized to store c levels of nodes for the modified additive tree (line 1). Then, we construct the groups formed by individual request $r_a \in R$ whose schedule consists of its source and destination for level 1 of the modified additive tree (lines 2-3). For the construction of the feasible groups of the remaining levels RG_l, we generate each node in level l by traversing and merging pairs of parent nodes' groups RG_{l-1} (lines 4-11). During constructing the nodes in level l, only groups G with l requests that satisfy Lemma IV.1 are considered (line 7). For each feasible group G, we search for the request $r_b \in G$ with the maximum shareability in SG (line 8). The new schedule S of G is generated by inserting r_b into its parent node's schedule S' using the linear insertion method (line 9). If the generated schedule is valid for G, we store the new group with its maintained schedule (G, S) into RG_l (lines 10-11).

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

Proof. Condition (a) is proved by Lemma 2 in [33] and condition (b) is derived from Observation 2 in Section III-A. □

Example 2. *Consider the example in Figure 1. We first initialize the groups composed of a single request with corresponding schedule. Next, we merge pairs of 1-size groups to create new schedules for child nodes. In the shareability graph SG in Figure 1(b), since $deg(r_2) > deg(r_3)$, we insert r_2 into the schedule of group $\{r_3\}$, forming $\{r_3, r_2\}$. The linear insertion method's schedule is optimal for groups of size 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 feasible schedule for r_4, r_1, all groups containing $\{r_1, r_4\}$ are pruned in subsequent steps. We then 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 $deg(r_2) > deg(r_1) = deg(r_3)$. As we cannot find any valid group G which $|G| \geq 4$, we end the group building process and got the final result in Figure 4.*

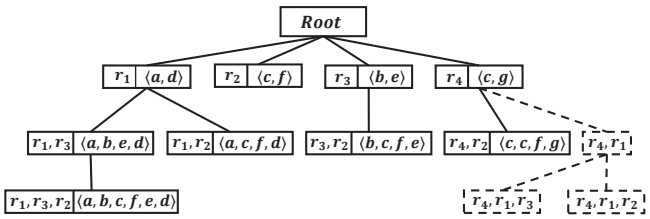


Fig. 4: Grouping Tree in Example 2.

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

C. SARD Algorithm

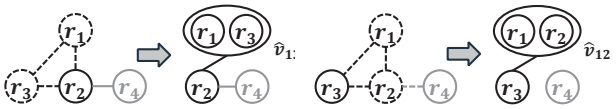
With the grouping algorithm in Section IV-B, we propose *SARD*, a two-phase matching algorithm for matching between vehicles and requests in this section. The intuition of SARD is to greedily maintain the shareability or connectivity of nodes in the shareability graph, increasing the likelihood of requests sharing with others in the final assignment. In SARD, requests initially propose to *worse* vehicles that result in higher travel costs, giving more initiative to vehicles on selecting groups of requests. Each vehicle v_j then greedily accepts a group of proposed requests with the smallest *shareability loss*, while rejected requests propose to better vehicles in later rounds. The proposal and acceptance phases are iteratively conducted until no request will propose. We start by defining shareability loss, then develop theoretical analysis to support our design of SARD.

To evaluate the effect of assigning a request group on the shareability of the remaining graph, we introduce a substitution operation to replace a k-clique G_i in shareability graph SG with a supernode $\hat{v}_i$. After we substitute a supernode $\hat{v}_i$ for a k-clique G_i, an edge connects 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 graph. Then, we define shareability loss as:

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

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

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



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

Fig. 5: An Illustration Example of Shareability Loss

Example 3. *With the shareability graph in Figure 1(b), we illustrate the idea of shareability loss in Figure 5, assuming r_4 is unavailable. In Figure 5(a), we aim to merge r_1 and r_3 into a supernode as follows. We first remove the edges incident to r_1 and r_3. Since r_1, r_2 and r_3 form a 3-clique in the original graph, there should be a shareable relation between r_2 and the supernode $\hat{v}_{13} = \{r_1, r_3\}$. Thus, we add a new edge between r_2 and $\hat{v}_{13}$. Overall, if we substitute $\hat{v}_{13}$ for $\{r_1, r_3\}$, the shareability loss is $SLoss(\{r_1, r_3\}) = 3-1 = 2$. In Figure 5(b), we try to merge r_1 and r_2 into a supernode $\hat{v}_{12} = \{r_1, r_2\}$. Similarly, four edges incident to r_1 and r_2 are removed, and a new edge between $\hat{v}_{12}$ and r_3 is built. Then, the shareability loss is $SLoss(\{r_1, r_2\}) = 4 - 1 = 3$ here. Therefore, substituting $\{r_1, r_3\}$ is more structure-friendly than substituting $\{r_1, r_2\}$.*

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

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

Proof. Observation 2 in Section III-A shows that all valid groups during assignment form a k-clique in the shareability graph. Therefore, we model maximizing sharing probability as a clique partition problem [48], aiming to find the minimum number of cliques to cover the graph, ensuring each node appears in precisely one clique. In the worst case, each node forms a 1-clique, resulting in a sharing probability of 0 (no requests can share). Intuitively, the less clique we use to cover the shareability graph, the higher the sharing probability is. Although Observation 2 provides a necessary but not sufficient condition for a clique to be shareable, cliques in the network still significantly boost the potential probability of requests being shareable. Hence, in the following proof, we illustrate the help of the shareability loss on the sharing probability by minimizing the number of cliques. Moreover, since vehicle capacity limits group size to k, we consider the problem of partitioning the shareability graph into a minimum number of cliques no larger than k. In [49], J. Bhasker *et al.* presented an optimal upper bound (shown in equation 6) for the clique partition problem evaluated with the number of nodes n and edges e for the graph.

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

In addition, we observe that the degrees of most riders in the shareability graph are relatively small, consistent with a power-law distribution. For analytical tractability, we assume the node degree in the shareability graph as a random variable δ following the power-law distribution with exponent η, which complementary cumulative distribution is shown as $\Pr(\delta \geq x) = ax^{-\eta}$. For a shareability graph SG with n nodes and the degree of its every node follows the power-low distribution with exponent η, Janson *et al.* [50] revealed that the size of the largest clique $\omega(SG(n,\eta))$ in the graph SG is a constant with $\eta > 2$. However, in the heavy-tail distribution, $\omega(SG(n,\eta))$ grows with $n^{1-\eta/2}$ (see equation 7).

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

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

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

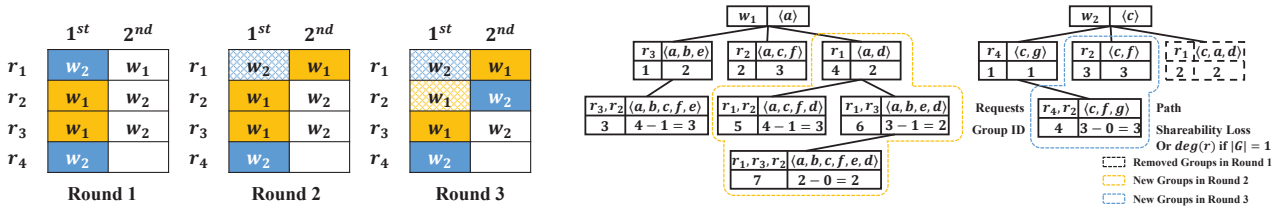
The higher the number of edges e in the shareability graph SG is, the lower the upper bound θ'_{upper} of the number of clique partitions is. In conclusion, substituting a supernode for the group of nodes with the lowest edge loss will keep the most edges left in the shareability graph SG, improving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

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

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

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

With Theorems IV.1 and IV.2, we design our SARD, whose pseudo code is shown in Algorithm 3. Firstly, we maintain a current working set R_p (line 1) containing all available requests until the current timestamps. When each batch arrives, we insert the requests in this batch into R_p (line 3) and process them along with the available (unmatched and unexpired) requests in the previous round (lines 2-17). Before starting the two-phase processing, we retrieve all candidate vehicles for each request r_a and store them in a priority queue $\mathcal{Q}_{r_a}$ in descending order of the increased utility cost for serving r_a



(a) Transformation of the Candidate Queue

(b) Grouping Trees for Vehicles

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

Algorithm 3: *SARD*

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

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

(lines 4-6). In the proposal phase (lines 8-10), the discarded requests in the previous round will propose to their current worst vehicle (i.e., the vehicle with the highest increase in travel cost for that request) (line 10). 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 12). Specifically, we replace the single request schedule in line 3 of Algorithm 1 with the schedule generated by inserting the request into the worker's current schedule. This helps filter out the groups that vehicle w_x cannot serve. With Theorem IV.1, we prioritize groups with minimal shareability loss and select such groups for vehicle w_x (line 13). Then, we put the discarded requests back to the working pool R_p (line 14), where $w_x.ac$ denotes the currently accepted requests of w_x. At the end of each acceptance phase, we assign the selected group to the vehicle and put the rejected requests back into the working set R_p for future proposals (lines 15-16). We repeat the proposal and acceptance phase until no requests remain. Finally, we remove expired requests from R_p and SG that cannot be completed due to exceeding the maximum waiting time at the end of each batch.

Example 4. *Let's examine the batch of requests in Example 1. Suppose two idle vehicles, w_1 and w_2, are located at a and c. We first create a candidate vehicle queue for $r_1 \sim r_4$. For example, the travel cost for w_1 and w_2 to serve r_1 is $cost(r_1)$ and $cost(r_1) + cost(c, a)$, respectively. Thus, the priority order in r_1's candidate queue is $\langle w_2, w_1 \rangle$. We can compute the candidate queue for the remaining requests, as shown in Figure 6(a). In the first round, r_2 and r_3 are added to w_1's candidate pool, while the remaining requests go to w_2's pool. Then, w_1 and w_2 enumerate all the groups by Algorithm 2 with the requests in their candidate pool (see Figure 6(b), excluding the colored dashed area). For w_1, we take $\{r_3, r_2\}$ as the temporal assignment as it's the only group with at least 2 members. Since there are no groups with at least 2 members in the first round of w_2, we prefer the group $\{r_4\}$, which has a lower degree by default as it leads to less shareability loss. Since r_1 is not accepted by w_2, it will propose to w_1 according to its candidate list in the second round. At this time, w_1's grouping tree will update to the groups shown in the Figure 6(b). Despite the same loss for G_6 and G_7, w_1 updates its best group to $\{r_1, r_3\}$ over $\{r_1, r_3, r_2\}$ due to a higher sharing ratio in $G_6 \frac{cost(P)}{\sum_{r \in SG} cost(r)}$. Therefore, r_2 will be discarded and propose in the third round, and w_2 will take $\{r_4, r_2\}$ by Theorem IV.2. Finally, w_1 and w_2 are assigned $\{r_1, r_3\}$ and $\{r_2, r_4\}$, respectively.*

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

V. Experimental Study

A. Experimental Settings

Data Sets. The road networks of *CHD* and *NYC* are retrieved as directed weighted graphs. The nodes in the graph are intersections, and the edges' weight represents the required travel time on average. There are 214,440 nodes and 466,330 edges in CHD road network (112,572 nodes and 300,425 edges in NYC road network). All algorithms utilize the hub labeling algorithm [51] with LRU cache [40] for shortest path queries on the road network.

We conduct experiments on three real datasets. The first dataset consists of requests collected by Didi in Chengdu, China (noted as *CHD*), published via its GAIA program [52]. The second one is the yellow and green taxi in New York,

TABLE II: Experimental Settings.

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

USA (noted as *NYC*) [53], which has been used as benchmarks in ridesharing studies [37], [41]. These datasets include each request's release time, source and destination coordinates, and the number of passengers. The third dataset is from a delivery service in Shanghai, China (referred to as *Cainiao*). Given space constraints, we put the content related to the third dataset in Appendix B in our technical report [42]. We focus on the days with the highest number of requests: November 18, 2016, for CHD (259K requests) and April 09, 2016, for NYC (250K requests). For requests with multiple riders, they are examined during the group enumeration in Algorithm 2. We set the deadline for request r_i as $d_i = t_i + \gamma \cdot cost(r_i)$, a commonly used configuration in many existing studies [40], [31], [41]. We set the maximum waiting time threshold for requests to 5 minutes follows previous work [23], which means $w_i = \min(5min, d_i - cost(r_i) - t_i)$. Figure 7 shows request source and destination distribution. Sources and destinations are mapped to road network nodes in advance. The initial positions of the vehicles are chosen randomly from the road network. Table II displays the parameter settings, with default values in bold. For simplicity, we set all vehicles to have the same capacity. The case of vehicles with different capacities is discussed in Appendix C of our technical report [42].

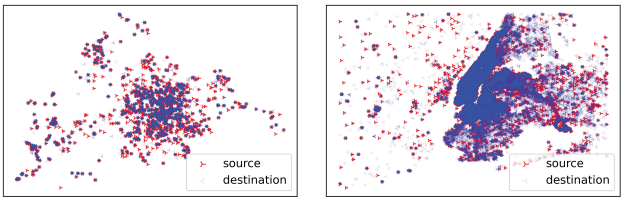
Approaches and Measurements. We compare our SARD with the following algorithms:

- ***pruneGDP [37].*** The request is added to the vehicle's current schedule in order and the vehicle with the smallest increase in distance to serve is chosen.
- ***RTV [27].*** It is in batch mode and obtains an allocation scheme with the least increased distance and most serving vehicles by linear programming.
- ***GAS [33].*** It enumerates vehicle-request combinations in random batch order, selecting the best group per vehicle based on total request length as profit.
- ***DARM+DPRS [54].*** It uses deep reinforcement learning to allocate idle vehicles to anticipated high-demand areas.
- ***TicketAssign+ [55].*** It proposes ticket locking on individual vehicles to enhance multiple-worker parallelism.

We present the following metrics of all algorithms.

- **Unified Cost.** The objective function of BDRP problem in Equation 3.
- **Service Rate.** The service rate is the ratio of assigned requests to the total number of requests.
- **Running Time.** We report the total running time of assigning all the tasks. The response time for a single request can be calculated by dividing the running time with the number of total requests (e.g., about 2 ms for each request in SARD).

We conduct experiments on a single server with Intel Xeon 4210R@2.40GHz CPU and 128 GB memory. All the algorithms are implemented in C++ and optimized by -O3 parameter. Besides, we use *glpk* [56] to solve the linear programming in the RTV algorithm, and all algorithms are implemented in a single thread except for TicketAssign+ .



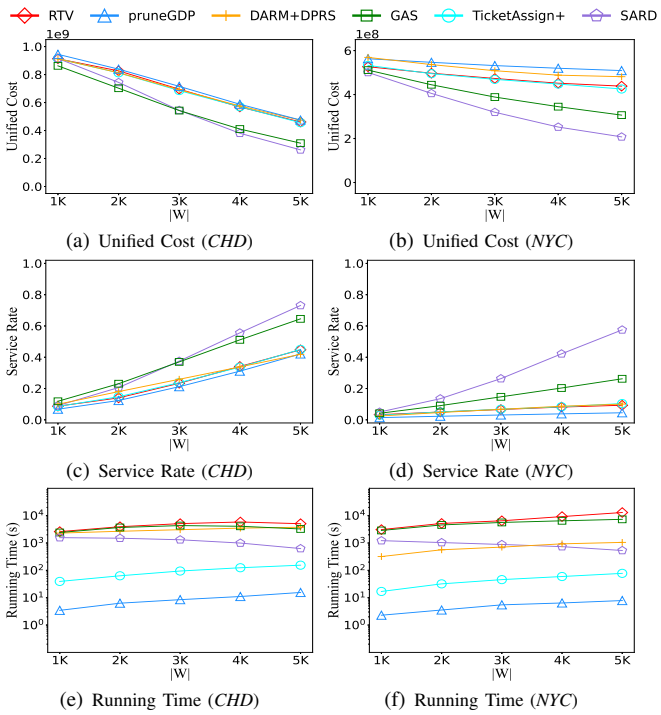
(a) Requests distribution (*CHD*) (b) Requests distribution (*NYC*)

Fig. 7: The distribution of datasets.

B. Experimental Results

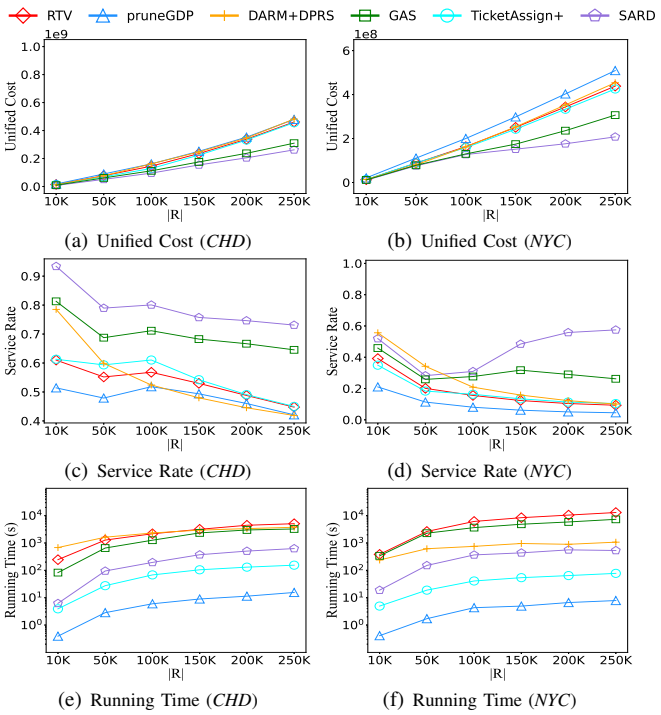
Effect of the number of vehicles. Figure 8 represents the results for 1K to 5K vehicles. For the unified cost, SARD and GAS are leading the other methods, and the margin between these two algorithms and the others are widening gradually. In the *CHD* dataset, the results of the two algorithms are very close, but SARD achieves an improvement of $2.09\% \sim 59.22\%$ compared with other methods in the *NYC* dataset. As for the service rate, since the penalty part of the unified cost caused by the rejected requests decreases when the algorithm achieves a higher overall service rate, the gap between the algorithms is consistent with unified cost trends. However, DARM+DPRS is an exception to this trend, as its scheduling of more idle vehicles increases travel time. TicketAssign+ achieves better service rates through simultaneous decision-making, thus escaping the local optima of the greedy algorithm. Our method improves service rates by up to 30.99% and 52.99% on two datasets. pruneGDP and TicketAssign+ excel in running time due to the efficiency of linear insertion, while TicketAssign+ experiences reduced speed as a result of vehicle resource contention . RTV, GAS, and SARD are slower than pruneGDP. Specifically, SARD achieves a speedup ratio up to $7.12\times$ and $23.5\times$ than RTV and GAS on two datasets, respectively. In addition, the running time of SARD even decreases with the increase of vehicle number, which is mainly due to the decrease in the number of propose-acceptance rounds. With more vehicles, fewer riders will conflict, and thus fewer rounds of the proposal are needed. This decreases the average proposed riders per vehicle, shortening the acceptance stage. DARM+DPRS showing more competitive speed on *NYC* but approaching batch-based algorithm runtime on *CHD* due to its larger state space.

Effect of the number of requests. Figure 9 shows that as requests increase from 10K to 250K, the unified costs of all algorithms grow. Meanwhile, SARD and GAS achieve smaller unified costs than other methods when the number of requests becomes larger. For service rate, SARD improves the service rate by $41.97\% \sim 52.99\%$ at most compared with pruneGDP. Besides, compared with the state-of-art batch-based method GAS, SARD achieves up to 12.09% and 31.27% higher service rates on the two datasets, respectively. DARM+DPRS demonstrates the advantages of predictive scheduling when the request volume is very small, but fails to handle a larger



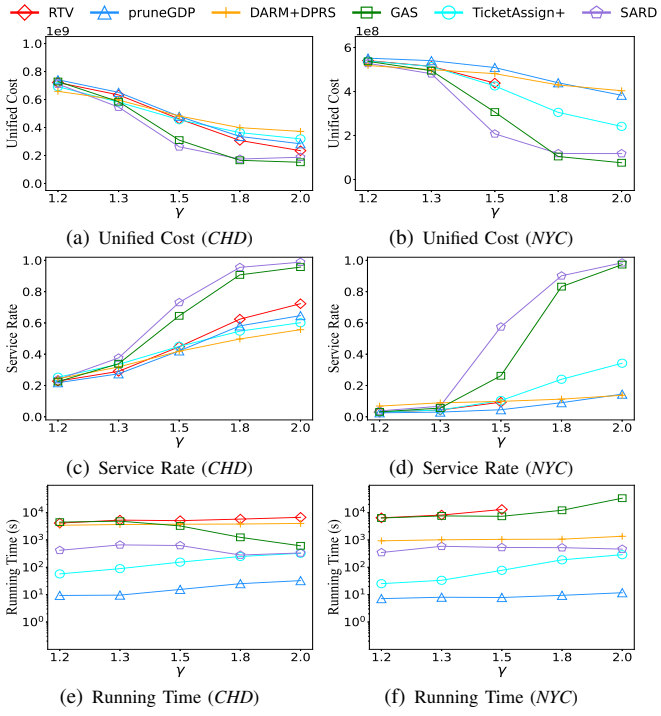
(a) Unified Cost (*CHD*) (b) Unified Cost (*NYC*)
(c) Service Rate (*CHD*) (d) Service Rate (*NYC*)
(e) Running Time (*CHD*) (f) Running Time (*NYC*)

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



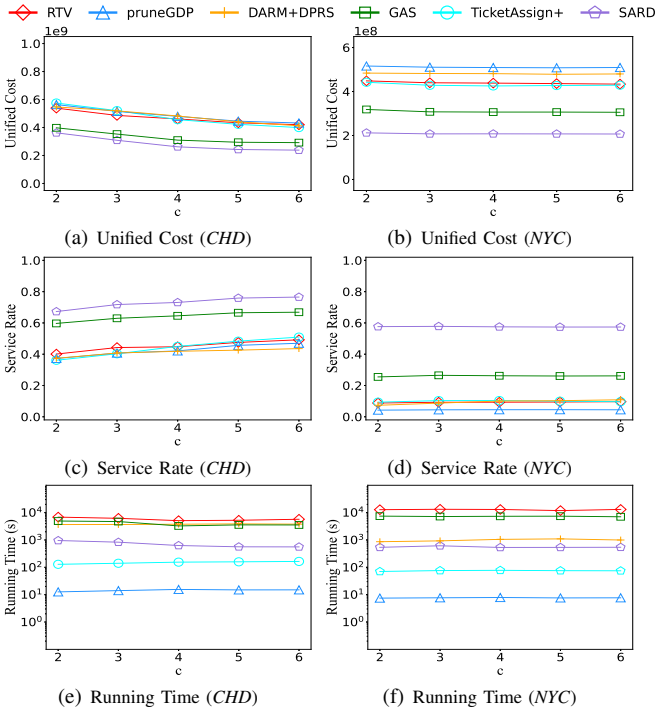
(a) Unified Cost (*CHD*) (b) Unified Cost (*NYC*)
(c) Service Rate (*CHD*) (d) Service Rate (*NYC*)
(e) Running Time (*CHD*) (f) Running Time (*NYC*)

Fig. 9: Performance of Varying $|R|$.



(a) Unified Cost (*CHD*) (b) Unified Cost (*NYC*)
(c) Service Rate (*CHD*) (d) Service Rate (*NYC*)
(e) Running Time (*CHD*) (f) Running Time (*NYC*)

Fig. 10: Performance of Varying γ.

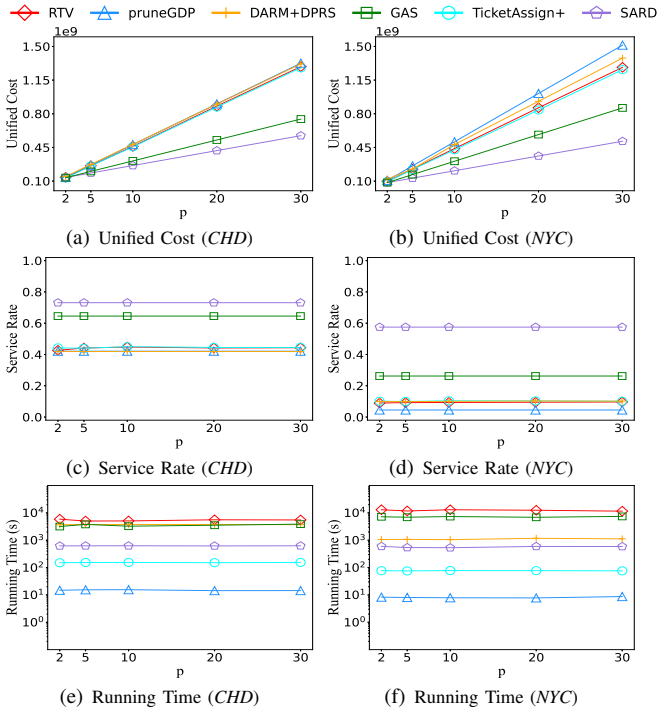


(a) Unified Cost (*CHD*) (b) Unified Cost (*NYC*)
(c) Service Rate (*CHD*) (d) Service Rate (*NYC*)
(e) Running Time (*CHD*) (f) Running Time (*NYC*)

Fig. 11: Performance of Varying c.

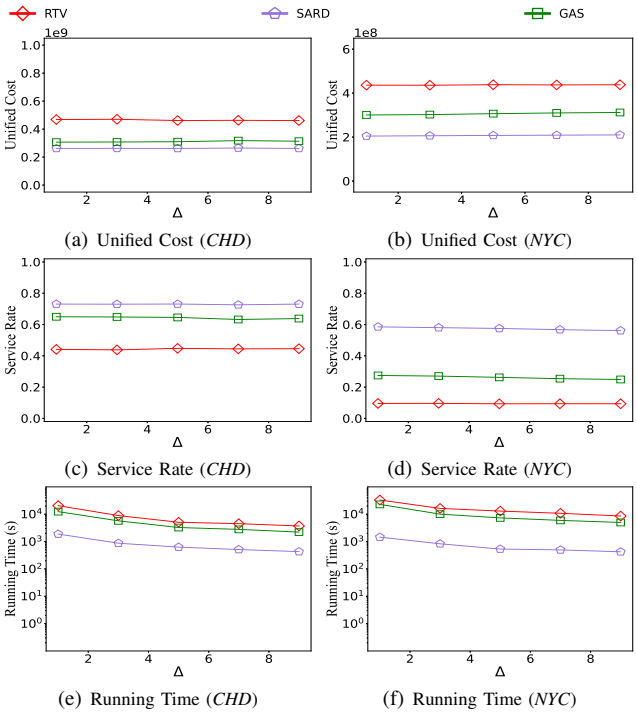
number of requests. All traditional algorithms initially exhibit a decline in service rate due to insufficient vehicles as requests increase. Subsequently, the service rate grows as the increased number of requests improves the probability of sharing rides with other requests, then more requests can be served. In terms of running time, insertion-based methods are faster. Among batch-based methods (SARD, RTV and GAS), SARD is $4.21\times \sim 38.6\times$ faster than GAS and RTV.

Effect of deadline. Figure 10 presents the impact of varying request deadlines by changing the deadline parameter γ from 1.2 to 2.0. With a strict deadline of $\gamma = 1.2$, SARD's service rate is similar to existing algorithms. This is due to a significant reduction in the number of candidate vehicles for each request, making it challenging to achieve noticeable performance improvements through grouping strategies in batch mode. However, as the deadline increases, the superiority of SARD and GAS gradually becomes apparent. TicketAssign+ increases contention with higher deadlines, boosting



(a) Unified Cost (*CHD*) (b) Unified Cost (*NYC*)
(c) Service Rate (*CHD*) (d) Service Rate (*NYC*)
(e) Running Time (*CHD*) (f) Running Time (*NYC*)

Fig. 12: Performance of Varying p_r.



(a) Unified Cost (*CHD*) (b) Unified Cost (*NYC*)
(c) Service Rate (*CHD*) (d) Service Rate (*NYC*)
(e) Running Time (*CHD*) (f) Running Time (*NYC*)

Fig. 13: Performance of Varying Δ.

service rate while increasing the gap in runtime compared to pruneGDP on the *NYC* dataset When the deadline is $1.8\times$, SARD's service rate exceeds 90%, up to 37.42% higher than that of other algorithms on the *CHD* dataset. The superiority of SARD is more evident in the *NYC* dataset, where its service rate is up to 83.98% higher than that of pruneGDP. GAS operates less efficiently on the *NYC* dataset with $\gamma = 2.0$, mainly because there are more request combinations as the deadline is relaxed. Additionally, GAS considers all combinations of requests and schedules for almost every vehicle. In SARD, the requests are proposed to different vehicles more decentralized in each round and only go to the next round of enumeration after being discarded. Thus, the time cost for the combination enumeration of each vehicle is significantly reduced in SARD, benefiting from our "proposal-acceptance" execution strategy. RTV results for $\gamma \geq 1.8$ on the *NYC* dataset are not presented because the constraints of RTV-Graph exceed the limit of *glpk* [56]. SARD achieves the best unified cost, saving up to 48.03% and 73.16% compared to others on two datasets. For the running time, SARD is $1.83\times$ to $72.68\times$ faster than RTV and GAS.

Effect of vehicle's capacity constraint. Figure 16 illustrates the results of varying vehicle capacity from 2 to 6. SARD and GAS save at least 30.87% in unified cost compared to other algorithms. For service rate, SARD is the best of all tested algorithms ($7.58\% \sim 53.39\%$ higher service rate than other tested algorithms). As for running time, SARD is also the fastest among batch methods (RTV, GAS, SARD), being $5.17\times \sim 17.52\times$ faster than RTV and GAS.

Effect of penalty. Figure 12 represents the effect of varying the penalty coefficient from 2 to 30. Most methods' service rates are unaffected by the penalty change. Since pruneGDP, TicketAssign+, DARM+DPRS, GAS and SARD take distance, group profit, and shareability loss as indicators in greedy strategies in the assignment phase, the penalty coefficient only affect their scores in the unified cost. However, RTV utilized the penalty coefficient in its linear programming (LP) constraint matrix, affecting LP results only when the penalty is small. For large penalties, the effect is negligible. Unified cost in all algorithms is proportional to the penalty coefficient. SARD leads in both datasets, with a service rate increase of $8.55\% \sim 52.99\%$. Similarly, since the penalty coefficient does not affect the assignment phase, the execution time remains stable across datasets.

Effect of the batching period. Figure 13 represents the results of varying the batching period from 1 to 9 seconds for batch-based methods. The varying batch time affects the batch-based algorithms in terms of unified cost and service rate. SARD performs the best with the varying the batching period, which saves up to 44.18% and 53.08% in unified cost and increases the service rate by up to 29.2% and 48.96% in two datasets, respectively. For the running time, since the number of execution rounds will decrease when the period of each batch increases, the running times of these methods decrease. SARD is $5.17\times \sim 24.47\times$ faster than RTV and GAS.

Effects of Angle pruning strategy. Table III shows the effect of the Angle pruning strategies proposed in Section III-B (parameters are in default values in Table II). We note the method without pruning strategies as *SARD*, with the Angle pruning as *SARD-O*. SARD-O saves up to 7.3% of the shortest path queries and saves 5.2% of the total running time on the two datasets compared with SARD. Besides, it has almost no

TABLE III: Performance of Pruning Strategies.

City	Method	Unified Cost	Service Rate	#Shortest Path Queries	Time (s)
CHD	*SARD*	263,682K	72.82%	2,971,425K	699.8
	SARD-O	260,641K	73.34%	2,839,734K	662.0
NYC	*SARD*	206,754K	57.64%	1,373,047K	770.6
	SARD-O	206,470K	57.83%	1,270,116K	750.6

harm on the service rate and unified cost.

Memory consumption. Figure 14 shows the memory usage of tested traditional algorithms under the default parameters. The online mode algorithms follow the first-come-first-serve mode and use less memory, while parallelization brings additional overhead of maintaining a lock for each worker. In contrast, the batch mode algorithms need additional storage to store the combinations of requests in each batch (e.g., RTV-Graph for RTV, the additive index for GAS, shareability graph for SARD). Since RTV rely on an integer linear program, the memory usage in RTV is more than twice that in GAS and SARD. Moreover, SARD and GAS have similar costs for storing undirected, unweighted shareability graphs.

Discussion. (1) **Running Time**. Online methods pruneGDP and TicketAssign+ have a complexity of $O(n^2)$, making them the fastest in the experiments. The running time of the reinforcement learning method DARM+DPRS hinges primarily on the size of its scheduled state space and the complexity of its predictive model structure. RTV, GAS, and our SARD are batch-based. 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, notated as n^c. GAS's complexity is $O(T_s + mT_c)$, with T_s for building an additive index (as $O(n^c)$), and T_c for greedily searching the additive index (as O(n^c)). Our SARD needs $O(cn^2 + mn^c)$, where n is the numbers of requests. Simplified, their complexities are $O(m^c n^2 c)$, $O(mn^c)$, and $O(mn^c)$, respectively. GAS and SARD outperform RTV. Unlike GAS, our SARD enumerates combinations only among proposed requests per vehicle, resulting in smaller n than GAS. Despite multiple propose-acceptance rounds, SARD's small n makes it much faster than GAS. (2) **DataSet**. In the *NYC* dataset, requests per unit time are about double those of *CHD*. Hence, in the same batch time, algorithms using combination enumeration (i.e., SARD and GAS) can get more candidate schedules and perform better in *NYC*. In addition, *NYC*'s more compact road network (with only half the nodes) and concentrated requests increase sharing opportunities. Thus, SARD performs better in *NYC* than *CHD* in most experiments. DARM+DPRS faces a greater challenge on the *CHD* dataset due to its larger state space. (3) **Scalability**. For SARD's scalability, multi-threading can speed up the Shareability Graph building and acceptance stage as each vehicle decides independently. In practice, SARD can be implemented with streaming distribution computation engines (e.g., Apache Flink [57]) and apply a tumbling window strategy to the distribution environment.

Summary of the experimental study:

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

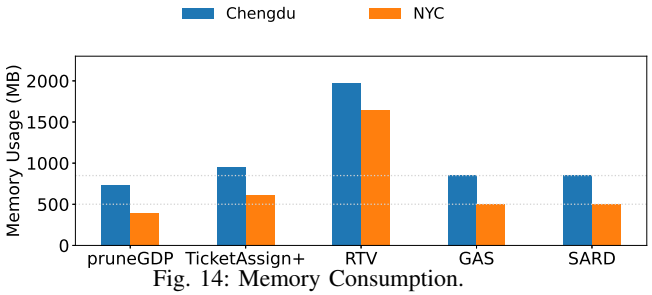


Fig. 14: Memory Consumption.

VI. Related Work

Ridesharing research dates back to the dial-a-ride problem [58]. Ridesharing services plan routes for each vehicle to fulfill requests while optimizing various objectives like minimizing travel time [27], [6], [7], maximizing served requests [3], [4], or both [5]. Liu *et al.* [29] proposed probabilistic routing for taxis to encounter suitable offline passengers. Existing works are either online-based [37], [59], [36], [31], [41] or batch-based [33], [60], [34], [27] depending on whether the requests are known in advance.

The *insertion* operation is a commonly used core operation [37], [59], [36], [41], [7] in online-based methods. It inserts the origin-destination pair of a new request into the current route. Chen *et al.* [30] utilized grid indexing to filter vehicles and applied deadline constraints pruning strategy to reduce shortest path distance calculations when inserting requests into the Kinetic Tree [7]. Tong *et al.* [37] minimized travel time with a dynamic programming method and proposed a linear-time insertion, applied in [59]. Xu *et al.* [36] sped up insertion to linear time by leveraging a segment-based DP algorithm and Fenwick tree. Wang *et al.* [41] enhanced the results by considering insertion effects and demand prediction.

In batch-based methods, requests are grouped by similarity before assignment. Zheng *et al.* [34] used bipartite matching to group similar riders and drivers. Zeng *et al.* [33] developed an index called the *additive tree* to simplify the process and used randomization techniques to improve the approximation ratio. Alonso-Mora *et al.* [27] explored cliques in the pairwise shareability graph and incrementally computed the optimal assignment using an integer linear program (ILP). However, none of these methods prioritize batches by taking advantage of the shareable relationships between requests.

VII. Conclusion

This paper studies the dynamic ridesharing problem with a batch-based processing strategy. We introduce a graph-based shareability graph to reveal the sharing relationship between requests in each batch. With the structure information of the shareability graph, we measure the shareability loss of each request group. We also propose a heuristic algorithm, *SARD*, with a two-phase "proposal-acceptance" strategy. Experiments show that our method provides a better service rate, lower cost, and shorter running time compared to state-of-the-art batch-based methods [33], [27] on real datasets.

REFERENCES

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

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

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

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

[5] H. Luo, Z. Bao, F. M. Choudhury, and J. S. Culpepper, "Dynamic ridesharing in peak travel periods," *CoRR*, vol. abs/2004.02570, 2020.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[23] P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti, "Quantifying the benefits of vehicle pooling with shareability networks," *Proceedings of the National Academy of Sciences*, vol. 111, no. 37, pp. 13290–13294, 2014.

[24] T.-Y. Ma, "On-demand dynamic bi-/multi-modal ride-sharing using optimal passenger-vehicle assignments," in *2017 IEEE International Conference on Environment and Electrical Engineering and 2017 IEEE Industrial and Commercial Power Systems Europe (EEEIC/I&CPS Europe)*, pp. 1–5, IEEE, 2017.

[25] Y. Chen and L. Wang, "P-ride: A shareability prediction based framework in ridesharing," *Electronics*, vol. 11, no. 7, p. 1164, 2022.

[26] J. Lin, S. Sasidharan, S. Ma, and O. Wolfson, "A model of multimodal ridesharing and its analysis," in *2016 17th IEEE International Conference on Mobile Data Management (MDM)*, vol. 1, pp. 164–173, IEEE, 2016.

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

[28] R. Kucharski and O. Cats, "Exact matching of attractive shared rides (exmas) for system-wide strategic evaluations," *Transportation Research Part B: Methodological*, vol. 139, pp. 285–310, 2020.

[29] Z. Liu, Z. Gong, J. Li, and K. Wu, "Mobility-aware dynamic taxi ridesharing," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 961–972, 2020.

[30] L. Chen, Q. Zhong, X. Xiao, Y. Gao, P. Jin, and C. S. Jensen, "Price-and-time-aware dynamic ridesharing," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1061–1072, 2018.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[46] C. Wang, Y. Song, Y. Wei, G. Fan, H. Jin, and F. Zhang, "Towards minimum fleet for ridesharing-aware mobility-on-demand systems," in *40th IEEE Conference on Computer Communications, INFOCOM 2021, Vancouver, BC, Canada, May 10-13, 2021*, pp. 1–10, IEEE, 2021.

[47] H. Zhang and J. Zhao, "Mobility sharing as a preference matching problem," *IEEE Trans. Intell. Transp. Syst.*, vol. 20, no. 7, pp. 2584–2592, 2019.

[48] R. M. Karp, "Reducibility among combinatorial problems," in *Proceedings of a symposium on the Complexity of Computer Computations* (R. E. Miller and J. W. Thatcher, eds.), The IBM Research Symposia Series, pp. 85–103, Plenum Press, New York, 1972.

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

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

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

[52] "[online] GAIA ." https://outreach.didichuxing.com/research/opendata/, 2016.

[53] "NYC dataset." https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[54] M. Haliem, G. Mani, V. Aggarwal, and B. K. Bhargava, "A distributed model-free ride-sharing approach for joint matching, pricing, and dispatching using deep reinforcement learning," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, pp. 7931–7942, 2020.

[55] J. J. Pan and G. Li, "Fast and scalable ridesharing search," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 11, pp. 6159–6170, 2024.

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

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

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

[59] Y. Zeng, Y. Tong, and L. Chen, "Last-mile delivery made practical," *Proceedings of the VLDB Endowment*, 2019.

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

[61] "Cainiao." https://www.cainiao.com/.

[62] "Cainiao dataset." https://huggingface.co/datasets/Cainiao-AI/LaDe-D.

APPENDIX

A. Symbols and Notations

We summarize the symbols and notations used in this paper in Table IV.

TABLE IV: Symbols and Notations.

Symbol	Description
$R = \{r_i\}$	Set of requests.
$W = \{w_j\}$	Set of vehicles.
s_i, e_i	Source and destination of request r_i.
t_i, d_i	Release time and deadline of request r_i.
n_i	Number of riders in request r_i.
c_j	Capacity of vehicle w_j.
S_j	Schedule of vehicle w_j.
$buf(o_k)$	Maximum allowed detour time of way-point o_k.
$ddl(o_k)$	Deadline of way-point o_k.
$cost(o_k, o_l)$	Travel cost between way-points o_k and o_l.
p_r	Penalty coefficient for unassigned requests.
γ	Deadline parameter.
$\mathcal{P}$	Batch of requests.
Δ	Batch time.
θ	Angle between two requests.
δ	Angle threshold for pruning.
SG	Sharability graph.
$SLoss(G_i)$	Shareability loss of group G_i.

B. Experimental Study on Cainiao Dataset

We conduct comprehensive experiments using the Cainiao [61] dataset, a real-world dataset from a prominent delivery platform in Shanghai, China, published by LaDe [62]. This dataset encompasses millions of packages and provides detailed daily information on tasks and workers from the Cainiao platform throughout 2021. Unlike taxi datasets, the delivery dataset exhibits a more dispersed request distribution and features more generous deadlines, allowing for greater routing flexibility. Due to insufficient training data, we only report the results of traditional algorithms here. Table V presents the parameter settings, with default values in bold.

TABLE V: Experimental Settings.

Parameters	Values
the number, n, of requests	50K, 75K, **100K**, 125K, 150K
the number, m, of vehicles	3K, 3.5K, 4K, 4.5K, **5K**
the capacity of vehicles c	2, 3, **4**, 5, 6
the deadline parameter γ	1.8, 1.9, **2.0**, 2.1, 2.2
the penalty coefficient p_r	2, 5, **10**, 20, 30
the batching time Δ (s)	1, 3, **5**, 7, 9
the variance σ	**0.0**, 0.5, 1.0, 1.5, 2.0

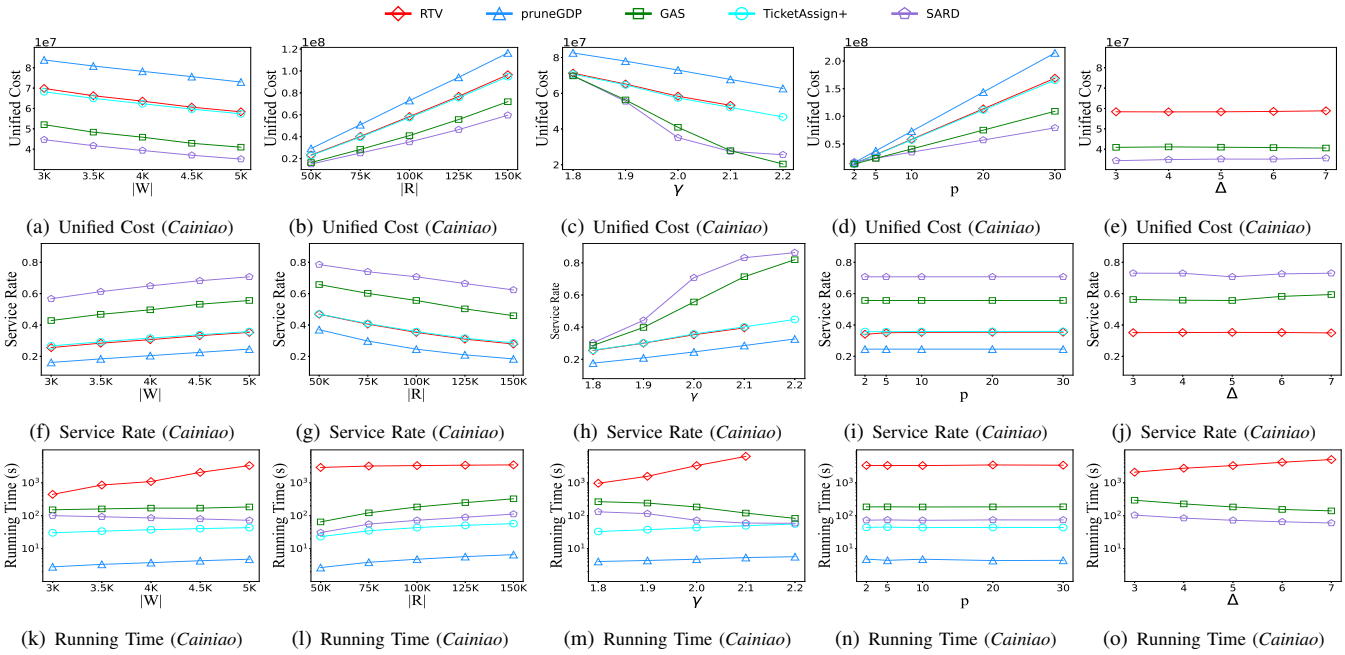
Effect of the number of vehicles. The first column in Figure 15 shows results as the number of vehicles varies from 3K to 5K. For unified cost, SARD and GAS outperform other methods, with SARD achieving a $14.16\% \sim 51.85\%$ improvement over other methods on the *Cainiao* dataset. Our method also demonstrates up to a 46.16% improvement in service rate on the *Cainiao* dataset compared to existing methods. Compared to pruneGDP, TicketAssign+ enhances service rates through concurrent decision-making processes, effectively mitigating the local optima challenges typically associated with greedy algorithms. All algorithms exhibit trends in unified cost that correspond consistently with their service rates. In terms of running time, pruneGDP excels as a result of its efficient linear insertion, while TicketAssign+ experiences increased running time because of vehicle contention. RTV, GAS, and SARD are slower than pruneGDP. Notably, SARD achieves up to a $46.01\times$ speedup compared to RTV and GAS on the *Cainiao* dataset. SARD's running time exhibits an inverse correlation with the number of vehicles. This enhancement in efficiency can be attributed to a reduction in propose-acceptance iterations. As the number of available vehicles (m) increases, competition among riders for individual vehicles decreases, resulting in fewer necessary proposal rounds.

Effect of the number of requests. The second column in Figure 15 illustrates that as requests increase from 50K to 150K, the unified costs of all algorithms grow. SARD and GAS achieve lower unified costs compared to other methods as the number of requests increases. Regarding service rate, SARD outperforms pruneGDP by up to 41.59%. Moreover, SARD achieves up to 16.44% higher service rates than the state-of-the-art batch-based method GAS on the *Cainiao* dataset. In terms of running time, insertion-based methods prove faster. Among batch-based methods (SARD, RTV, and GAS), SARD demonstrates $2.91\times \sim 31.15\times$ faster performance than GAS and RTV.

Effect of deadline. The third column in Figure 15 illustrates the impact of varying request deadlines by adjusting the deadline parameter γ from 1.8 to 2.2. With a strict deadline of $\gamma = 1.8$, SARD's service rate is comparable to existing algorithms. This stems from a significant reduction in candidate vehicles for each request, making it difficult to achieve notable performance improvements through batch mode grouping strategies. However, as the deadline extends, the superiority of SARD and GAS becomes increasingly evident. At a deadline of $2.1\times$, SARD's service rate surpasses 80%, up to 54.60% higher than other algorithms on the *Cainiao* dataset. RTV results for $\gamma = 2.2$ on the *Cainiao* dataset are omitted due to RTV-Graph constraints exceeding the *glpk* [56] limit. SARD achieves the best unified cost, saving up to 59.12% compared to others on the *Cainiao* dataset. Regarding running time, SARD is $1.41\times$ to $106.19\times$ faster than RTV and GAS.

Effect of penalty. The fourth column in Figure 15 illustrates the effect of varying the penalty coefficient from 2 to 30. Most methods' service rates remain unaffected by this change. pruneGDP, TicketAssign+, GAS, and SARD incorporate distance, group profit, and shareability loss as indicators in their greedy assignment strategies. Consequently, the penalty coefficient only influences their unified cost scores. In contrast, RTV integrates the penalty coefficient into its linear programming (LP) constraint matrix. This affects LP results solely when the penalty is small; for larger penalties, the impact becomes negligible. The unified cost for all algorithms shows a proportional relationship to the penalty coefficient. SARD outperforms in *Cainiao* datasets, achieving a service rate increase of $15.10\% \sim 46.16\%$. As the penalty coefficient doesn't influence the assignment phase, execution times remain consistent across datasets.

Effect of the batching period. The final column in Figure 15 illustrates the impact of varying the batching period from 3 to 7 seconds for batch-based methods. This variation in batch

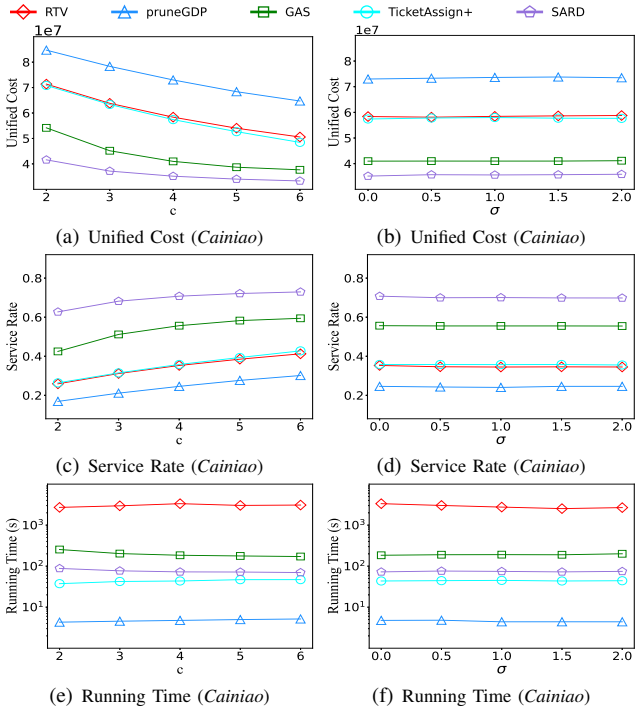


(a) Unified Cost (*Cainiao*) (b) Unified Cost (*Cainiao*) (c) Unified Cost (*Cainiao*) (d) Unified Cost (*Cainiao*) (e) Unified Cost (*Cainiao*)

(f) Service Rate (*Cainiao*) (g) Service Rate (*Cainiao*) (h) Service Rate (*Cainiao*) (i) Service Rate (*Cainiao*) (j) Service Rate (*Cainiao*)

(k) Running Time (*Cainiao*) (l) Running Time (*Cainiao*) (m) Running Time (*Cainiao*) (n) Running Time (*Cainiao*) (o) Running Time (*Cainiao*)

Fig. 15: Performance of Cainiao Dataset when Varying $|W|$, $|R|$, γ, p_r and Δ.

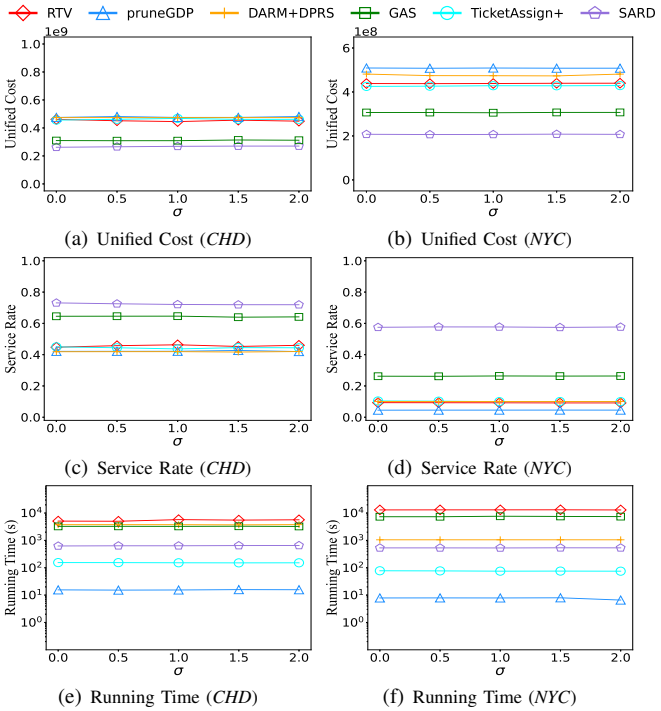


(a) Unified Cost (*Cainiao*) (b) Unified Cost (*Cainiao*)

(c) Service Rate (*Cainiao*) (d) Service Rate (*Cainiao*)

(e) Running Time (*Cainiao*) (f) Running Time (*Cainiao*)

Fig. 16: Performance of Varying c and σ.



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

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

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

Fig. 17: Performance of Varying σ.

time influences the unified cost and service rate of batch-based algorithms. SARD demonstrates superior performance across varying batching periods, achieving up to 39.48% reduction in unified cost and a 38.1% increase in service rate on the *Cainiao* dataset. Regarding execution time, as the duration of each batch increases, the number of execution rounds decreases, resulting in reduced running times for most methods. However, RTV experiences increased time costs due to a higher number of matches. Notably, SARD exhibits $2.33\times \sim 84.82\times$ faster performance compared to RTV and GAS.

Effects of Angle pruning strategy. Table VI shows the effect of the Angle pruning strategies proposed in Section III-B (parameters are in default values in Table V). We note the method without pruning strategies as *SARD*, with the Angle pruning as *SARD-O*. SARD-O saves up to 41.9% of the shortest path queries and saves 33.9% of the total running time on the *Cainiao* dataset compared with SARD. Besides, it has almost no harm on the service rate and unified cost.

Memory consumption. Figure 18 illustrates the memory usage of tested traditional algorithms under default parameters. Online-based algorithms, following a first-come-first-serve ap-

TABLE VI: Performance of Pruning Strategies.

City	Method	Unified Cost	Service Rate	#Shortest Path Queries	Time (s)
Cainiao	*SARD*	351,22K	70.84%	782,827K	109.870
	SARD-O	351,58K	70.75%	454,887K	72.543

proach, consume less memory. However, parallelization incurs additional overhead due to the need to maintain a lock for each worker. In contrast, batch-based algorithms require extra storage to hold request combinations for each batch. For instance, RTV employs an RTV-Graph, GAS uses an additive index, and SARD utilizes a shareability graph. RTV's reliance on an integer linear program results in memory usage more than 20 times that of GAS and SARD. The latter two algorithms use similar memory for their shareability graphs.

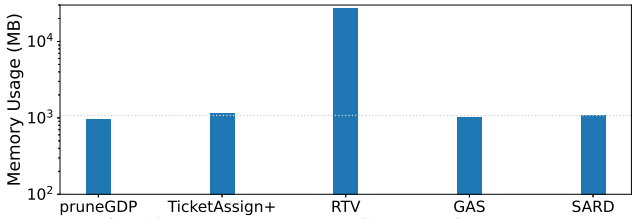


Fig. 18: *Cainiao* Memory Consumption.

Effect of vehicle's capacity constraint. The first column in Figure 16 presents the outcomes of adjusting vehicle capacity from 2 to 6. SARD and GAS demonstrate superior performance, achieving a minimum of 22.26% reduction in unified cost compared to alternative algorithms. All evaluated algorithms exhibited enhanced service quality due to expanded sharing opportunities. In terms of service rate, SARD outperforms all other tested algorithms, achieving a $13.51\% \sim 42.80\%$ higher service rate. Regarding execution time, SARD emerges as the most efficient among batch methods (RTV, GAS, SARD), executing $2.45\times \sim 44.19\times$ faster than RTV and GAS.

C. Experimental Study on Vehicle Capacity Distribution

To evaluate the efficacy of ride-sharing under scenarios where different vehicles can have different capacities, we set the variance parameter σ as shown in Table V. This parameter generates diverse vehicle capacity distributions adhering to a normal distribution with a mean of 4 and varying variances. Our previous default configuration is considered to have a variance of 0. We conduct tests on our *Cainiao*, *NYC*, and *CHD* datasets, as shown in Figure 16 and 17. All algorithms remain stable across the three metrics, indicating that the distribution of different vehicle capacities had a negligible impact on ride-sharing quality.