Online Ridesharing with Meeting Points

Jiachuan Wang †, Peng Cheng *, Libin Zheng †, Lei Chen †, Xuemin Lin #,*
†*The Hong Kong University of Science and Technology, Hong Kong, China*
jwangey@cse.ust.hk, lzhengab@cse.ust.hk, leichen@cse.ust.hk
**East China Normal University, Shanghai, China*
pcheng@sei.ecnu.edu.cn
#*The University of New South Wales, Australia*
lxue@cse.unsw.edu.au

***Abstract*—With the development of the online platform, ridesharing becomes a popular commuting mode in our daily life. Specifically, dynamically arriving riders post their origins and destinations, then the platform assigns drivers to serve them. In ridesharing, different groups of riders can be served by one driver if their trips can share common routes. Recently, many ridesharing companies (e.g., Didi and Uber) further propose a new mode, namely "ridesharing with meeting points". Specifically, with a short walking distance but less payment, riders can be picked up and dropped off around their origins and destinations, respectively. In addition, meeting points enables more flexible routing for its driver, which can potentially improve the global profit of the system. In this paper, we first formally define the Meeting-Point-based Online Ridesharing Problem (MORP). We prove that MORP is NP-hard and there is no polynomial-time deterministic algorithm with a constant competitive ratio for it. To pre-select the "meeting points", we introduce a novel meeting point candidates selection algorithm. We further propose a hierarchical meeting-point oriented graph (HMPO graph) to accelerate the whole ridesharing process. Finally, we propose a novel algorithm, namely SMDB, to solve MORP. Extensive experiments on real and synthetic datasets validate the effectiveness and efficiency of our algorithms.**

I. Introduction

Nowadays, on-demand ridesharing becomes important in civil commuting services. Together with online platforms (e.g., DiDi [1]), ridesharing surpasses traditional taxi services with more energy saved, less air pollution, and lower cost [32].

In *online ridesharing*, riders arrive dynamically. Platforms need to deal with them immediately for different objectives, including maxmizing the number of served riders [12], [14], [25], [30], [40], minimizing the total travel distance [8], [20], [21], [23], [26], [28], [29], [30], [34], or maximizing the unified revenue [9], [10], [36].

Ridesharing allows one driver to serve more than one group of riders simultaneously. The route of a driver is a sequence of pick-up/drop-off points. Given a set of drivers and riders, *route planning* is to design and update routes every time a rider arrives. A key operation, called *insertion*, shows great effectiveness and efficiency for solving online ridesharing problem [26], [23], [34], [28], [30], [40], [11], [12], [36]. It tries to insert a newly coming rider's origin and destination into a driver's route without changing the order of his/her current sequence of pick-up/drop-off points.

However, due to the complex topology of the city road network, some locations are spatially close to each other but hard to access for vehicles. Especially, if two locations are only connected by a Pedestrian Street where vehicles cannot go through, a short walking could greatly reduce the travel cost of the assigned vehicle. To deal with the case, *meeting points (MP for short)* are introduced as alternative locations for pick-up/drop-off locations of riders [31]. As shown in Figure 1. a rider r at location **A** wants to go to location **C**. The nearby roads are directed roads. A driver w at location **B** is assigned to serve r. Then, the shortest route for w is represented in blue dashed arrow lines. If r can move a short distance, for example, to location **D** (i.e., a MP), w can serve r through a much shorter route displayed in the green line.

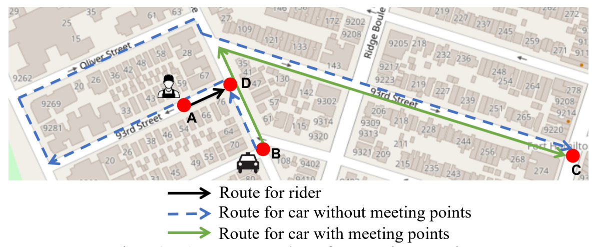


Fig. 1: An Example of Meeting Points

In a recent work [41], the authors utilize meeting points to improve the results of offline ridesharing problems, whose methods however only can handle up to 40 riders/vehicles. Thus, it is not practical for online applications (e.g., Uber and DiDi) with hundreds of riders/vehicles every several seconds. Existing studies also investigate the strategies to properly select MPs with online surveys [13], [15]. In industry, Uber recently offers Express POOL to encourage riders to walk to Express spots (meeting points) for efficient routing [5]. Nevertheless, Uber Express POOL only schedules the route for each vehicle when there are sharable ride-requests to group with, otherwise, the rider need wait until other shareable riders come. In addition, the MPs in Uber Express Pool are similar to the stops of buses for nearby riders to come together and thus not flexible [6]. For the example in Figure 1, Uber Express POOL will not assign driver w to pick up rider r until another rider r' appears close to point D (i.e., the selected pick-up stop). In summary, to the best of our knowledge, in the existing research works, there is no solution for the *online* ride-sharing services boosted with flexible MPs.

To improve the efficiency of the online flexible ridesharing, we define a new problem called *meeting-point-based online route planning* problem (MORP), which utilizes MPs as flexible pick-ups or drop-offs to reduce the travel cost of vehicles and shorten the waiting time of riders. Based on existing studies [9], [36], we prove that the MORP problem is NP-hard and has no deterministic algorithms with a constant competitive ratio, thus intractable.

To solve the MORP problem, we first propose a novel method to evaluate the importance of each vertex offline , which guides us to select a *meeting point candidate set* for each vertex to accelerate the MP search. In addition, we design a structure, namely hierarchical meeting-point oriented graph (HMPO graph), to assign hierarchical order to vertices for better assignments. We further devise a new insertion algorithm, namely SMDB, based on HMPO, which can solve MORP effectively and efficiently.

Here we summarize our main contributions:

- We formulate the online route planning problem with MPs mathematically, namely MORP. We prove that it is NP-hard and has no algorithm with constant competitive ratio in Section III.
- With observations and analyses, we propose a heuristic algorithm to select MP candidates for riders in Section IV, which is based on a unified cost function considering the travel cost from additional walking.
- We propose a novel hierarchical structure of the road network, namely hierarchical meeting-point oriented (HMPO) graph, to fasten the solution for MORP in Section V.
- Based on the HMPO graph, we propose an effective and efficient insertor, namely SMDB, to handle the requests in MORP in Section VI.
- We conduct extensive experiments on synthetic and real data sets to show the efficiency and effectiveness of SMDB in Section VII.

II. Background and Related Works

Route planning for ridesharing, which has been widely studied in recent years, is a variant of the dial-a-ride problem (DARP) proposed in 1975 [37], [38]. Traditional DARP problems usually have additional restrictions, such as limiting the drivers to start from/return to depot(s) and serve all the requests [22], [19]. These settings lead to small scale datasets with near-optimal solutions. In comparison, route planning for ridesharing is more applicable in the real world, which applies to hundreds of thousands of requests and tens of thousands of drivers with locations distributed over large scale road network [9], [10], [23], [36]. Realistic revenue and serving cost can be designed as objectives to meet the requirement of ridesharing platforms [9], [10], [36], [42]. A common setting for the serving cost is a unified score based on distance/time cost of driving and penalty of rejecting riders. One can further extend the unified cost to an application-specific one, such as maximizing the score combined with complicated social utilities from both workers and requests [11], [16]. Using meeting point results in additional costs such as walking, which is handled with a unified cost function.

Besides, real-world ridesharing services require solutions for online instead of off-line mode. Efficient heuristic methods are developed for route planning without information of future workers and requests in advance [9], [10], [12], [23], [26], [28], [34], [40]. With large scale dataset and requirement for real-time response, a commonly used operator called insertion shows good performance for route planning [11], [12], [23], [26], [24], [27], [28], [29], [30], [34], [36]. Insertion greatly reduces the search space of possible new routes to serve each rider from $O(N!)$ to $O(N^2)$. Tong *et al.* further reduce its time complexity to linear time using dynamic programming [36], [35]. We adapt the linear insertion for our MORP problem as baseline and further propose a more effective insertor based on a new graph structure.

As an effective way to improve ridesharing experience, meeting points (MPs) are used in online hailing companies, such as Didi and Uber. Stiglic *et al.* [31] first introduce the concept of "meeting points" to give alternatives to pick up and drop off riders. They devise a heuristic algorithm for meeting-point-based offline ridesharing problem. Zhao *et al.* [41] develop the mathematical model for the offline ridesharing problem and propose an integer linear programming model to solve it. In recent years, Uber had proposed Express Pool as an online ridesharing service, in which riders need to walk a little but get a discount. However, riders need to take more time to wait for assignments. On the other hand, Uber prefers to group passengers together with the same MPs, then pick up and drop off them like a bus with selectable stations, which has less flexibility [6]. In this paper, we focus on the online ridesharing problem with MPs, which needs to respond to requests immediately (e.g., within 5 seconds).

III. Problem Definition

A. Basic Notations

We use graph $G_c = \langle V_c, E_c \rangle$ to represent a road network for cars, where V_c and E_c indicate a set of vertices and a set of edges, respectively. Each edge, $(u, v) \in E_c(u, v \in V_c)$, is associated with a weight $t_c(u, v)$ indicating travel time for driving from vertex u to v through it. Similarly, graph $G_p = \langle V_p, E_p \rangle$ is used to represent a road network for passengers. Each edge $(u, v) \in E_p(u, v \in V_p)$ is weighed by $t_p(u, v)$ as its travel time for walking. For the two graphs, we denote the union of vertices as $V = V_c \cup V_p$. In the city network, passengers are more flexible. We set all the edges in V_p undirected according to the network of OSM [7]. In addition, usually for any edge (u, v), walking is slower than driving (e.g., $t_c(u, v) < t_p(u, v)$). We denote *path* as a sequence of vertices $\{v_1, v_2 \cdots, v_k\}$ with travel time $\sum_{i=1}^{k-1} t(v_i, v_{i+1})$. For each pair of vertices (u, v), we represent the time cost of its shortest path for cars and passengers as $SP_c(u, v)$ and $SP_p(u, v)$, respectively.

Definition 1 (Drivers)**.** Let $W = \{w_1, w_2, \cdots, w_n\}$ be a set of n drivers that can provide transportation services. Each driver w_i is defined as a tuple $w_i = \langle l_i, a_i \rangle$ with a current location l_i and a capacity limitation a_i.

At any time, the number of riders in a taxi of driver w_i must not exceed its capacity a_i.

Definition 2 (Requests)**.** Let $R = \{r_1, r_2, \cdots, r_m\}$ be a set of m requests. Each request r_j can be denoted by $r_j = \langle s_j, e_j, tr_j, tp_j, td_j, p_j, a_j, pi_j, de_j, wp_j, wd_j \rangle$ with its source location s_j, destination location e_j, release time tr_j, latest pick-up time tp_j, deadline td_j, rejection penalty p_j, and a capacity a_j. Once it is assigned, two vertices as pick-up point pi_j and drop-off point de_j will be recorded. The shortest time for a request to walk from source to pick-up point is represented as $wp_j = SP_p(s_j, pi_j)$ and from drop-off point to destination is denoted as $wd_j = SP_p(de_j, e_j)$.

A request r_j can be served by driver w_i only if: (a) w_i can arrive at pi_j after tr_j; (b) the remaining capacity of w_i is larger than a_j when he/she arrives at pi_j; and (c) w_i can pick r_j at pi_j before tp_j and deliver r_j at de_j before $td_j - wd_j$.

Note that in real-application, rejections are unavoidable for the "urgent" requests on a platform, especially at rush hours. The loss from rejecting r_j is denoted by penalty p_j. The penalty can be application-specific. Furthermore, we denote all the requests that are served by driver w_i as R_{w_i}. Then, $\hat{R} = \cup_{w_i \in W} R_{w_i}$ and $\bar{R} = R \backslash \hat{R}$ refer to the total served and unserved requests, respectively. To simplify, we will use r_j to indicate a request or a rider of a request without differentiation.

Definition 3 (Meeting Points)**.** For a request $r_j \in R_{w_i}$, the pick-up point $pi_j = u \in V_c \cap V_p$ denotes that driver w_i will pick up rider r_j at vertex u. The drop-off point $de_j = v \in V_c \cap V_p$ denotes that w_i will drop off r_j at vertex v. Pick-up and drop-off points are *meeting points (MP for short)*.

With MPs, we allow the drivers to flexibly pick up and drop off passengers. Traditional online ridesharing solutions only assign drivers to pick up a rider r_j from its source s_j and drop off r_j to its destination e_j. With MPs, the rider r_j can move a short distance to location pi_j and be picked up there by a driver. After being dropped off at location de_j, r_j walks to his/her destination e_j.

Definition 4 (Route)**.** The route of a driver w_i located at l_i is a sequence, $S_{w_i} = [l_i, l_{x_1}, l_{x_2}, \cdots, l_{x_k}]$, where each l_{x_k} is a pick-up or drop-off point of a request $r_j \in R_{w_i}$ and the driver will reach these locations in the order from 1 to k.

We call the vertices of a route as *stations*. Drivers move on shortest paths between stations. A feasible route satisfies: (a) $\forall r_j \in R_{w_i}$, its drop-off time of de_j is earlier than $td_j - wd_j$; (b) $\forall r_j \in R_{w_i}$, its pick-up point pi_j appears earlier than its drop-off point de_j in S_{w_i}; (c) The total capacity of undropped riders is no larger than the driver's capacity a_i at any time. We denote $S = \{S_{w_i} | w_i \in W\}$ as all the route plans.

Here we define $D(S_{w_i})$ as the shortest time to finish S_{w_i}:

$$D(S_{w_i}) = SP_c(l_i, l_{x_1}) + \sum_{k=1}^{|S_{w_i}|-2} SP_c(l_{x_k}, l_{x_{k+1}})$$

B. Meeting Points based Online Ridesharing

Definition 5 (Meeting Points based Online Ridesharing Problem, MORP)**.** Given transportation networks G_c for cars and G_p for passengers, a set of drivers W, a set of dynamically arriving requests R, a driving distance cost coefficient α, a walking distance cost coefficient β, MORP Problem is to find a set of routes $S = \{S_{w_i} | w_i \in W\}$ for all the drivers with the minimal unified cost:

$$UC(W, R) = \alpha \sum_{w_i \in W} D(S_{w_i}) + \beta \sum_{r_j \in \hat{R}} (wp_j + wd_j) + \sum_{r_j \in \bar{R}} p_j \quad (1)$$

which satisfies the following constraints: (i) Feasibility constraint: each driver is assigned with a feasible route; (ii) Non-undo constraint: if a request is assigned in a route, it cannot be canceled or assigned to another route; if it is rejected, it cannot be revoked.

C. Hardness Analysis

Lemma III.1. *The MORP problem is NP-hard.*

Proof. Due to space limitation, please refer to Appendix A in our technical report [3]. □

The Competitive Ratio (CR) is commonly used to analyze the online problem. CR is defined as the ratio between the result achieved by a given algorithm and the optimal result for the corresponding offline scenario. The existing work proves no constant CR to maximize the total revenue for basic route planning for shareable mobility problems with neither deterministic nor randomized algorithm [9], [36]. Here we have the following lemma for MORP problem.

Lemma III.2. *There is no randomized or deterministic algorithm guaranteeing constant CP for the MORP problem.*

Proof. For details, please refer to Appendix B in [3]. □

To solve the MORP problem, we first introduce a novel MP candidate selection algorithm to efficiently and effectively select the potential MPs. Based on the selected MPs, we further design a hierarchical meeting-point oriented (HMPO) graph to solve the MORP problem effectively. Finally, we propose a meeting-point-based insertion operator, namely SMDB, which prunes candidate drivers using the HMPO graph.

IV. Select Meeting Point Candidates

After Stiglic et al. [31] introduced the concept of "meeting points" (MP) to provide flexible pick-up and drop-off points for riders, many researchers aim to find an effective solution for ridesharing with MPs. Instead of searching for MPs every time a request arrives, which is time-consuming, we pre-select some vertices as candidates to serve their nearby vertices.

In this section, we first introduce the motivation of selecting MP candidates. Then, we propose a heuristic algorithm, Local-Flexibility-Filter, to select them.

A. Meeting Point Candidates

With the MPs, we have more choices for the pick-up and drop-off points. In real applications, riders' walking time is usually less than a constant (e.g., 4 minutes). We assume that on average, a vertex has K nearby vertices within the rider's maximum walking time. If we simply apply the traditional solutions to all pairs of MPs and find the optimal one, the time cost would be K^2 times as large as the time cost of the algorithms without MPs, which is unacceptable. On the other hand, some vertices are not convenient to access and thus unnecessary to check. Thus, we define *Meeting Point Candidates* as a pre-selected candidate set for convenient MPs. When we insert a rider according to its source and destination, we can quickly check their MP candidates instead of searching for MPs from scratch.

Definition 6. (Meeting Point Candidates) Given the road network for cars $G_c = \langle V_c, E_c \rangle$, for each vertex $u \in V_p$, its

MP Candidates is a set of vertices $MC(u) = \{v_1, v_2, \cdots\} \subseteq V_c \cap V_p$. For the MORP problem on G_c, we only select MPs for a vertex u from its $MC(u)$, that is, $\forall r_j \in \hat{R}, pi_j \in MC(s_j)$ and $de_j \in MC(e_j)$.

B. Meeting Point Candidate Selection

MP candidates should easily get to and conveniently reach other vertices. We introduce our Local-Flexibility-Filter algorithm to find the candidate sets $MC(\cdot)$ in two phases.

Vertices convenient for drivers. The first phase aims to find the vertices which are convenient for drivers, thus boosting transportation efficiency. To quantify the convenience, we introduce the *equivalent in/out cost* ECI/ECO for each vertex. When we insert a vertex u into a route between v_1 and v_2, its $ECI(u)$ indicates the average cost $SP_c(v_1, u)$ while $ECO(u)$ is for $SP_c(u, v_2)$.

As riders are usually assigned to nearby drivers, surrounding vertices are more important to estimate ECI/ECO. For each vertex, we define its *reference vertices* as those vertices with good indications to evaluate its convenience. Specifically, for a source vertex u, we select n_r nearest vertices on the car graph G_c as reference vertices. To estimate $ECO(u)$ of u, we collect its n_r nearest vertices as a set $n_o(u)$, and calculate the sum of the distances from u to these vertices as $ST(u) = \sum_{v \in n_o(u)} SP_c(u, v)$. Intuitively, we define the equivalent out cost as $ECO(u) = ST(u)/n_r$. Similarly, for $ECI(u)$ of u, we select another n_r reverse nearest vertices as $n_i(u)$ and calculate the sum of the distances from them to u as $ST'(u)$. Then, $ECI(u) = ST'(u)/n_r$ indicates the average cost of reaching u from other vertices. Here, n_r depends on the density of a road network and the speed of drivers.

Vertices convenient for riders. The second phase takes the walking convenience into account. Initially, for each vertex u, we select vertices $\{v_1, v_2, \cdots\}$ no farther than a maximal walking distance d_m. For each reachable vertex v_i, we calculate a serving-cost score $SCS(u, v_i)$ combining both walking distance and the equivalent in/out costs as $SCS(u, v_i) = \beta \cdot SP_p(u, v_i) + \alpha\,(ECI(u) + ECO(u))$, where α and β are the weight factors for driving and walking cost. Especially, each vertex u has the *SCS* score for itself: $SCS(u, u) = \beta \cdot SP_p(u, u) + \alpha\,(ECI(u) + ECO(u)) = \alpha\,(ECI(u) + ECO(u))$.

Size of $MC(\cdot)$. Large candidate set weaks the pruning effectiveness, while small candidate set may not contain good MPs. Thus, we need to decide the proper size of $MC(\cdot)$.

For a vertex u, all its candidates with a score higher than $SCS(u, u) + thr_{CS}$ are pruned, where thr_{CS} is a user-specified threshold. The intuition is that a vertex with a low average cost to serve a rider does not need alternatives. But a vertex with high SCS needs more choices to find a good MP.

We set an upper bound nc_m for the number of candidates of each vertex. If a vertex has more than nc_m candidates, we choose the top nc_m vertices according to their SCS.

Especially, for each vertex $u \in V_p/V_c$, $ECI(u) = ECO(u) = \infty$. For each vertex $u \in V_c/V_p$, $SP_p(u, \cdot) = \infty$. According to the definition of SCS, these two types of vertices will never be selected as MPs, that is, $MC(\cdot) \subseteq V_c \cap V_p$.

Algorithm 1: Local-Flexibility-Filter Algorithm

```
   Input: Graph G_c for cars and G_p for passengers, the number of
          reference vertices n_r, maximum walking distance d_m,
          threshold thr_CS for pruning candidates, maximum
          number of candidates nc_m
   Output: MP candidates MC for each vertex
1  Initialize integer lists ST, ST' with sizes of |V_c|.
2  Initialize set lists n_i and n_o with sizes of |V_c|.
3  Initialize dictionary list MC with size of |V_p|.
4  foreach vertex u ∈ V_c of G_c do
5      Find the n_r nearest vertices for u on G_c.
6      Calculate ST(u) and derive ECO(u).
7      Find the n_r reverse nearest vertices for u on G_c.
8      Calculate ST'(u) and derive ECI(u).
9  foreach vertex u ∈ V_p do
10     Find the accessible vertices, A_u, for u on G_p within d_m.
11     foreach vertex v_i ∈ A_u do
12         If SCS(u, v_i) ≤ SCS(u, u) + thr_CS, add it to u's MP
            candidates and only keep top nc_m points.
13     Add these candidate vertices into MC(u).
14 return MC
```

Now, we have the MP candidate set for each vertex. Once a request arrives, we find the candidate MPs only in the MP candidate sets of its source and destination.

Algorithm sketch. Algo. 1 Local-Flexibility-Filter Algorithm (LFF Algorithm for short) shows the detail of our candidate selection solution. As vertices in road network have bounded number of adjacent edges (usually no more than 5), the degree of a vertex can be regarded as $O(1)$. We will stop after finding n_r nearest vertices in line 5 and line 8, so there are n_r rounds to pop vertex and relax edge. The time complexity of lines 5-8 is $O(n_r \log(n_r))$. The total time complexity from lines 4 to 8 is thus $O(|V|\, n_r \log(n_r))$. We assume that given any vertex u, the maximum number of vertices that can reach u within d_m on G_p is n_w, line 10 would have a time complexity $O(n_w \log(n_w))$. Line 12 costs $O(\log(nc_m))$ by using a min-heap to keep top nc_m points. The lines 11-12 thus costs $O(n_w \log(nc_m))$. Line 13 costs $O(n_w)$. The total cost from line 9 to 13 is $O(|V|\, n_w(\log(n_w) + \log(nc_m)))$. Because maximal number of candidates is always smaller than the number of reachable vertices, that is, $nc_m < n_w$, the time complexity of LFF algorithm is $O(|V|\,(n_w \log(n_w) + n_r \log(n_r)))$.

V. Hierarchical Meeting-Point Oriented Graph

MPs enable alternative pick-up and drop-off points for each request, but also involve more calculations. For each vertex, K MP candidates results in K^2 times computational cost of the traditional process. In this section, we first discuss the properties of vertices in real-world data. Based on the mining results, we design a Hierarchical Meeting-Point Oriented graph (*HMPO graph*), which exploits the flexibility for higher profits but reduces additional computing costs.

A. Graph Analysis for Meeting Point

Take the road network of New York City on Open Street Map (OSM) [7] as an example, which contains 58189 vertices and 122337 edges. In this real-world road network, some vertices are "hotter" than the others, which have larger traffic flows and lower transition costs. Based on this intuition, we evaluate the original road graph and vertex-flexibility with methods in Section.IV. Based on the result in Section.IV, vertices can be ranked by $ECI(\cdot)+ECO(\cdot)$ as an indication of "hot". We find that with only 20% "hottest" vertices, more than 70% vertices are directly servable, which means each of them has at least one MP candidate point included in the "hottest" vertices. According to this observation, the MPs enable the drivers to fast serve most of their requests on "hot" roads. On the other hand, some vertices are inconvenient to drive in and out for drivers. For the riders with inconvenient origins or destinations, MPs can be their convenient alternatives. If we give the vertices a hierarchical order, the arrangement could be more effective. This motivates us to build a hierarchical graph, which gives an indication for optimal assignment with MPs and boosts the query on "hotter" vertices.

B. Structure of Hierarchical Meeting-Point Oriented Graph

We introduce a new structure, called Hierarchical Meeting-Point Oriented Graph (HMPO Graph), which gives the hierarchical order over the vertex set V and has 3 levels of vertices:

- **Core vertices** V_{co}. They are used as MPs frequently.
- **Defective vertices** V_{de}. They are inconvenient to access.
- **Sub-level vertices** V_{su}. The remaining vertices are classified as sub-level vertices.

The three sets of vertices form a partition of all vertices in G_c and G_p, that is, $V_{co} \cup V_{su} \cup V_{de} = V, V_{co} \cap V_{su} = \emptyset, V_{co} \cap V_{de} = \emptyset$, and $V_{su} \cap V_{de} = \emptyset$. We introduce an example below, which will be used to show the main steps of our algorithms to build a HMPO graph in this section.

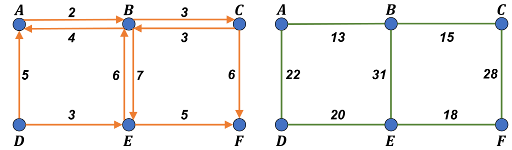
Example 1. *There are 6 vertices A to F in Figure 2. The graph for car is directed in Figure 2(a). The graph for passenger is undirected shown in Figure 2(b). Considering $n_r = 3$ nearby vertices, we derive $ECO(\cdot)$ and $ECI(\cdot)$ according to Section IV in Table I. With $\alpha = \beta = 1$, $nc_m = 3$, $d_m = 30$, and $thr_{CS} = 15$, we derive the MP candidates $MC(\cdot)$ shown in Table I.*

TABLE I: ECO and ECI with 3 nearby vertices, and $MC(\cdot)$

ID	A	B	C	D	E	F
$ECO(\cdot)$	5.33	4.67	5.33	5	6.67	∞
$ECI(\cdot)$	5.33	3.67	5.67	∞	6.33	6.33
$MC(\cdot)$	$\{A, B\}$	$\{B\}$	$\{B, C\}$	$\{A, D, E\}$	$\{E\}$	$\{C, E, F\}$

1) Defective Vertices: Defective vertices are inconvenient vertices for vehicles to access and will be eliminated. Based on the selection of MP candidates in Section IV, a vertex with extremely large ECO and ECI or has less than n_r nearest vertices is usually inconvenient for drivers to reach and leave. In MORP, we can substitute them with convenient MPs to improve the effectiveness and efficiency of the assignment. This motivates us to remove the vertices that are hard to access and construct a more concise graph.

Vertex removing cost. Once we remove a vertex u from a traditional graph, there are two kinds of additional costs while doing the shortest path query: (i) *the detour cost*, which occurs



(a) Graph for car G_c (b) Graph for passenger G_p

Fig. 2: Original Graph for car and passenger

Algorithm 2: Defective Vertices Selection Algorithm

Input: Graph $G_c = \langle V_c, E_c \rangle$ for cars, list ECO and ECI from Algo.1, MP candidates MC
Output: The set of defective vertices V_{de}
1 Sort the vertices in V_c in decreasing order of $ECO(v_i) + ECI(v_i)$
2 Initialize defective vertices $V_{de} = \emptyset$
3 Initialize reserved vertices $V_{re} = \emptyset$
4 **foreach** *Vertex* $u \in V_c$ **do**
5   **if** $u \in V_{re}$ *or* $MC(u) \subseteq V_{de}$ **then**
6   Continue
7   Put vertices that can directly reach to u into inV
8   Put vertices that can be directly reached from u into $outV$
9     $t_m = \max_{v \in inV} t_c(v, u) + \max_{v \in outV} t_c(u, v)$
10   Remove u and its related edges from G_c
11   **foreach** *Vertex* $v \in inV$ **do**
12   Put every vertex v' with distance $SP_c(v, v') \leq t_m$ into V_{tm}
13   **if** $outV \nsubseteq V_{tm}$ **then**
14   Put u with its related edges back to G_c
15   Check next vertex from Line 5
16   **while** *Vertex* $v' \in V_{tm}$ **do**
17   **if** $v' \in outV$ *and* $SP_c(v, v') > t_c(v, u) + t_c(u, v')$ **then**
18   Put u with its related edges back to G_c
19   Check next vertex from Line 5
20   Add u to V_{de} and add all vertices in $MC(u)$ to V_{re}
21 **return** V_{de}

if a query finds a path containing u. Removing u means that we need to find a new path without passing u. The new path is usually longer than the original one and results in a detour; (ii) *the inaccessibility cost*, which is from queries with u as origins or destinations. No path will exist after removing u.

With MP candidates for each vertex in G_p, we can get the walking cost directly without searching the original G_p. Thus, we only have shortest path queries on the graph G_c. We focus on G_c to make it concise and eliminate some defective vertices to improve the speed of shortest path query on it.

Note that pruning vertices for the traditional ridesharing problem leads to rejections for riders start or end there. But with MPs, the inaccessibility cost of pruning vertices can decrease from ∞ to a limited mixture of walking and driving costs. Hereby, with MPs, extracting defective vertices for faster assignment is much safer when we benefit from the efficiency. We propose a heuristic algorithm called Defective Vertices Selection Algorithm (*DVS algorithm* for short) to prune the defective vertices without error from the detour cost and maintain its accessibility in the meantime.

Algorithm sketch. The pseudo code of the DVS approach is shown in Algorithm 2. The main idea of the DVS approach is to ensure that: (a) every selected defective vertex will have at least one MP candidate existing in the rest vertices (i.e., core vertices or sub-level vertices); (b) the travel cost of any two vertices in G_c will not increase after removing the defective vertices and their related edges.

The implementation of the DVS approach (Algorithm 2) can be summarized in 3 steps:

(i) Sort $v_i \in V$ in descending order of $ECO(v_i) + ECI(v_i)$;

(ii) Initialize each vertex as unmarked, which means that it is not required by any defective vertices as MP;

(iii) pop vertices one-by-one. For each vertex u, check whether it is marked or its MP candidates $MC(u)$ are all in V_{de}. If so, pop the next vertex. Otherwise, remove it from G_c and record each vertex v which has an edge with u. If the edge starts from v (an edge comes to u), we add v into a set inV with the weight $t_c(v,u)$; if it starts from u, we add v into a set $outV$ with the weight $t_c(u,v)$. Save the sum of the largest in and out weights as t_m. Then we remove u with its adjacent edges from G_c. For each vertex $v \in inV$, we run the Dijstra algorithm from source v until the new reached vertex costs more than t_m. If any $v' \in outV$ is accessed with cost larger than $t_c(v,u)+t_c(u,v')$, we refuse to add u into V_{de} and check next one. If all vertices in $outV$ have been accessed, add u to V_{de} and mark vertices in $MC(u)$. u and its edges are recovered if we failed to add it to V_{de}.

A vertex u with a large ECO/ECI is hard to reach and leave, thus it has a high possibility to be isolated from the city center and the main street. Removing the defective vertices will have little influence on the connectivity of the road network. We sort the vertices according to their ECO and ECI at the beginning of DVS, which help us remove the unimportant vertices first as they affect fewer vertices, such that a more concise HMPO graph can be achieved.

Example 2. *Let us continue the setting in Example 1. Now we want to select the defective vertices among them. We first sort the vertices and initial their labels as unmarked. Then we check all the vertices one-by-one and show their status at each round with a row in Table II. Round 0 initializes flags as unmarked (u). Round 1 pops F and there is no node in inV. We add it to V_{de} and mark its candidates $MC(F)$ as m. In the table we use de to suggest that F is added to V_{de}. Round 2 adds D to V_{de} and updates its unmarked candidate(s) (A). Rounds 3 5 all pop marked vertices and fail to add a new vertex to V_{de}. Round 6 checks the B by first removing it from the graph with edges and recording its $inV = outV = \{A, C, E\}$ with $tm = 6+7 = 13$. Running Dijkstra algorithm from A results in ∞ cost and poping B is rejected. We mark the failed ones as f.*

Time Complexity. We assume that each vertex has an $O(1)$ degree, which is common in road networks. Line 1 runs in $O(N \log N)$ time, where $N = |V|$. Lines 7 and 8 collect adjacent vertices in $O(1)$ time as degree is in $O(1)$ scale. Assume that (i) the longest length-2 path in G_c costs t_{near}; (ii) given any vertex $u \in G_c$, the number of vertices that u can reach within cost t_{near} is no larger than σ_{near}, then the number of accessed vertices is bounded by σ_{near}. Thus, line 12 is a $O(\sigma_{near} \log \sigma_{near})$-time-complexity loop. All checking phases (lines 5, 13, 17) are implemented in hash index and cost $O(1)$. Loop from lines 11 to 19 and loop from lines 16 to 19 enumerate $O(1)$ vertices. Loop from line 4 to 20 enumerates $O(N)$ vertices. Thus, the overall time complexity is $O(N \cdot (\log N + \sigma_{near} \log \sigma_{near}))$.

TABLE II: Status of each vertex at each round

Round	F	D	E	C	A	B
0	u	u	u	u	u	u
1	de	u	m	m	u	u
2	de	de	m	m	m	u
3	de	de	f	m	m	u
4	de	de	f	f	m	u
5	de	de	f	f	f	u
6	de	de	f	f	f	f

Lemma V.1. *Removing all vertices selected by the DVS algorithm from G_c with their edges leads to no detour cost.*

Proof. For details, please refer to Appendix C in [3]. □

In addition, though defective vertices are inaccessible for drivers on the new graph, we still maintain the ability to serve influenced requests with lemma below.

Lemma V.2. *$\forall u \in V$ is accessible after removing vertices selected by the DVS algorithm from G_c with MPs.*

Proof. For details, please refer to Appendix D in [3]. □

2) Core Vertices: In the city network, the highway provides quick travel channel for drivers. People in nearby places usually drive to the highway and go along it, then finally turn to another road for the destination. For the traditional ridesharing problem, one can use this property by organizing more segments of each rider's trip along highways. However, drivers still need to leave the highway for pick-up and drop-off in the middle. MP naturally handles this issue and takes full advantage of it. Hence, we are motivated to make use of the fast and request-concentrated road segment for the MORP problem. The main idea is to find a group of core vertices V_{co} as a backbone of the whole graph. With the help of the MP, we can arrange most of or even all the requests with core vertices as pick-up/drop-off points, then drivers can drive along highways at a high speed in most time. In addition, the possibility of serving new nearby riders will increase, which creates more profits. In short, little additional walking distance results in a high overall profit. We define the Core Vertices Selection Problem:

Definition 7. Given the car graph G with its selected defective vertices V_{de}, the Core Vertices Selection Problem is to find a set of core vertices $V_{co} \subseteq V - V_{de}$ with the minimum size $|V_{co}|$, such that

(i) V_{co} is a k-skip cover of the updated graph $G_{c'} = G_c - V_{de}$

(ii) ϵ percent of vertices in V_p has at least one vertex $u \in V_{co}$ as its MP candidate.

Attribute **(i)** involves the concept of the ***k*-skip cover**, which is proposed by Tao et al. [33]. Given a graph $G(V, E)$, we call a set $V^* \subseteq V$ a k-skip cover if for any shortest path SP on G with exactly k vertices, there is at least one vertex $u \in SP$ satisfying $u \in V^*$. In general, for any shortest path SP in G, vertices of $SP \cap V^*$ succinctly describes SP by sampling the vertices in SP with a rate of at least $\frac{1}{k}$. Such a sub-path out of the whole path is called a k-skip shortest path. In many applications, such as electronic map presentation, given all vertices are unnecessary and the k-skip shortest path gives a good skeleton of it. The study further shows that answering k-skip queries is significantly faster than finding the original shortest paths.

In our study, core vertices take the main responsibility to serve most of the requests. A great number of queries can be boosted with the k-skip cover structure. MPs enable us to assign many more requests along the road network skeleton captured by V^*, which can fully exploit the advantage of it than the traditional ridesharing problem. Funke et al. [17] further generalize the work of [33] by constructing k-skip path cover for all paths instead of only shortest paths. Besides, they devise a new way of constructing a smaller size of k-skip path cover. Both of their works tend to remove the vertices with a bad connection to their neighbors and keep the vertices with greater potential to be the component of more shortest paths. This underlying rule coincides with our requirement for core vertices as a highway-skeleton.

Attributes **(ii)** assures that most of the requests can be assigned with core vertices as MPs. For fast computation on the skeleton after further preprocessing, the size of core vertices should be minimized. Note that when we find MP candidates for each vertex, there is an upper bound nc_m for the number of candidates. Hereby, to satisfy the attribute **(ii)**, the problem can be converted to the partial set cover problem with low-frequency items by (i) constructing candidate serving set MS to indicate the set of servable vertices for each vertex $u \in V - V_{de}$, that is, for any pair of vertex $u, v \in V, u \in MC(v)$ *iff* $v \in MS(u)$. As an inverse relationship of MC, we can convert the candidate MP set by scanning each vertex $u \in V$ and for each $v \in MC(u)$, add u to $MS(v)$. This process is linear; (ii) the universe $\mathcal{U}$ is defined to be the set of all the passenger-accessible-vertices V_p; (iii) find the smallest sub-collection of vertices $\{u_1, u_2, \cdots\} \subseteq V - V_{de}$ that the union of their serving sets covers at least ϵ of the universe, that is, $|\cup_k MS(u_k)| \geq \epsilon \cdot |V_p|$.

The partial set cover problem is an NPC problem. Luckily, in our setting, each element (vertex) is in at most nc_m candidate serving sets as the times of adding a vertex u to some $MS(\cdot)$ equals to the number of u's MP candidates $|MC(u)| \leq nc_m$. Such a case is called a low-frequency system. There exists a solution to approximate the optimum within a factor nc_m using LP relaxation in polynomial time [18].

Formally, it can be formulated as the following integer linear program (ILP):

$$\begin{aligned}
\min \quad & \sum_{u \in V - V_{de}} \delta_u \\
\text{s.t.} \quad & \phi_v + \sum_{u: v \in MS(u)} \delta_u \geq 1 && \forall v \in V_p \\
& \sum_{v \in V_p} \phi_v \leq (1 - \epsilon)\, |V_p| \\
& \delta_u \in \{0, 1\} && \forall u \in V - V_{de}, \\
& \phi_v \in \{0, 1\} && \forall v \in V_p,
\end{aligned} \tag{2}$$

where $\delta_u = 1$ iff vertex u is chosen to serve its candidate serving set $MS(u)$. The binary variable $\phi_v = 1$ iff vertex v is not served by any selected vertex.

Algorithm 3: Core Vertices Selection Algorithm

Input: All the vertices V and defective vertices V_{de}, list ECO and ECI from Algo.1, MP candidates MC
Output: The set of core vertices V_{co}

```
1 Initialize candidate serving set for each u ∈ V − V_de as ∅.
2 foreach v ∈ V_p do
3     foreach u ∈ MC(v) do
4         Add v to MS(u);
5 Solve the partial set cover problem using the algorithm in [18],
    where the weights of sets are substituded with ECO + ECI in
    its sorting step. Finally, record its cost cost_u of choosing each
    MS(u) as the highest cost set. Denote its output as V'_co.
6 Initialize the core vertices V_co = V − V_de.
7 Sort the vertices u ∈ V − V'_co − V_de in decreasing order of
    cost_u. Check them one-by-one and remove a vertex from V_co if
    the V_co is still a k-skip cover.
8 Sort and check the vertices u ∈ V'_co in decreasing order of
    cost_u. Remove a vertex v from V_co if V_co − {v} is still a
    k-skip cover and the MSs of V_co − {v} cover ϵ · |V_p| vertices.
9 return V_co
```

The corresponding LP relaxation can be derived by substituting the constraints $\delta_u \in \{0, 1\}$ by $\delta_u \geq 0, \forall u \in V - V_{de}$ and $\phi_v \in \{0, 1\}$ by $\phi_v \geq 0, \forall v \in V_p$. Then the problem is transferred to a linear Program LP. After deriving the dual LP of it, [18] iteratively chooses each set to be the highest cost set and obtains a feasible cover with the primal-dual stage. Finally, it chooses the solution with minimum cost.

However, to guarantee attribute (**i**) simultaneously, we cannot apply this algorithm directly to our problem. Both of the two attributes require the set of chosen vertices to have a good "coverage" of other vertices, that is, each of them can serve many nearby vertices. Motivated by this observation, we introduce a heuristic algorithm 3, *CVS*, to solve this problem.

Algorithm sketch. We construct it with three steps:

(1) In lines 1-4, we first construct candidate serving sets $MS(\cdot)$. Then we obtain the partial set cover solution $V'_{co} \in V - V_{de}$ satisfying attribute (i) using [18]. As each set (a vertex with its MS in our setting) has the same cost 1, we sort each vertex in increasing order of their $ECO + ECI'$ at the sorting step of their algorithm. During the process, we record the cost $cost_u$ of choosing each candidate serving set $MS(u)$ to be the highest cost set. Return their output V'_{co}.

(2) Initialize the core vertex set $V_{co} = V - V_{de}$. Sort the vertices $\forall u \in V - V_{de} - V'_{co}$ according to $cost_u$ in decreasing order. Intuitively, this can be treated as how bad the "quantity" each vertex u with its serving set $MS(u)$ to be included. The higher its value, the worse a vertex is. We check each vertex $u \in V - V_{de} - V'_{co}$ in order and remove it from V_{co} if the constraint of k-skip shortest path is not violated.

(3) Sort the vertices $\forall u \in V'_{co}$ according to $cost_u$ in decreasing order. Pop each vertex $u \in V'_{co}$ in order and check whether the MSs of remaining vertices cover ϵ vertices and satisfy a k-skip cover over $G_c - V_{de}$. If so, remove it.

Example 3. *Let us go back to the setting in Example 1. After selecting F and D to V_{de}, we select the core vertices among the rest of vertices. We set $\epsilon = 80\%$ to guarantee that no* *fewer than $\epsilon|V| = 4.8$ vertices are covered. $MS(\cdot)$ and the result of partial set cover are shown in Table III. The final cover is $V'_{co} = \{B, E\}$ with cost 2. To maintain a 2-skip cover, we initialize $V_{co} = \{A, B, C, E\}$ and iteratively check each vertex $u \in V - V_{de} - V'_{co} = \{A, C\}$. As none of these removals violates the 2-skip cover, both of them are removed from V_{co}. Removing B or E violates attribute (ii) and we finally output $V_{co} = \{B, E\}$.*

TABLE III: Candidate serving set and partial set covers.

ID	A	B	C	E
$MS(\cdot)$	$\{A, D\}$	$\{A, B, C\}$	$\{C, F\}$	$\{D, E, F\}$
Partial set cover	$None$	$None$	$\{C, A, B\}$	$\{E, B\}$
Cost	∞	∞	3	2

Time Complexity. Lines 1-4 construct $MS(\cdot)$ in $O(|V| \cdot nc_m)$ time. Line 5 costs time as same as the solution in [18] costs, which is $O(|V|^2)$. In lines 6 to 8, checking k-skip cover constraint costs $O(\bar{\sigma}_{k-1} \log \bar{\sigma}_{k-1})$, where $\bar{\sigma}_{k-1}$ is the average number of $(k-1)$-hop neighbors of the vertices in V, which grows linearly with $|V|$ [17]. Using a hashset to record a copy of MC can help us check the servable vertices in $O(1)$ time. With an $O(|V|)$ loop, lines 6 to 8 cost $O(|V|^2 \log |V|)$ time. So the overall time complexity is $O(|V|^2 \log |V|)$.

The size of V_{de} satisfies the following lemma:

Lemma V.3. *Assume that we have N vertices in total, with M set as optimal solution for the attribute* (ii)*, the upper bound of the size of core vertex set is $\sigma(k) = \max(\frac{N}{k} log \frac{N}{k}, nc_m \cdot M)$.*

Proof. For details, please refer to Appendix E in [3]. □

Finally, as V_{co}, V_{su}, and V_{de} is a partition of V, it is trivial that $V_{su} = V - V_{co} - V_{de}$.

C. Construction of HMOG Gragh

After obtaining the three levels of vertices, we use them to construct the hierarchical graph $G_h(V, E_h)$. In this subsection, we concentrate on the formulation of E_h.

First, for the defective vertices, we "discard" them so there is no edge for drivers to come nor leave them. As requests with origins and destinations among V_{de} are always servable with MPs. Besides, there is no additional time cost for any shortest path after removing V_{de}. Drivers can move along more convenient routes without these vertices, leading to a high potential profit against the limited discarding cost.

Recall that V_{co} is a k-skip cover of $V - V_{de}$. Based on that, a special graph structure can be constructed for fast query [33]. To be more specific, a new graph $G^* = (V_{co}, E_{cc})$ together with 3 edge sets E_{cs}, E_{sc}, and E_{ss} are derived. All the shortest distance queries can be answered by G^* efficiently.

Core vertices serve as the skeleton of the original road network. Borrowing the definition in the previous work [33], for each $u \in V_{co}$, one can find its *k-SKIP NEIGHBORS* and build *super-edges*. Any query result between these core vertices on the new graph is the same as that on the original graph. Query start from or aim at sub-level vertices can be answered by temporally extending the graph with super-edges between sub-level vertex and its nearby core vertices.

Here we summarize the construction steps. (1) Instead of the k-SKIP NEIGHBOR of u, $N_k(u)$, we can find a superset $M_k(u)$ of $N_k(u)$ efficiently according to [33]. (2) We build super-edge, which is weighted by the shortest path distance

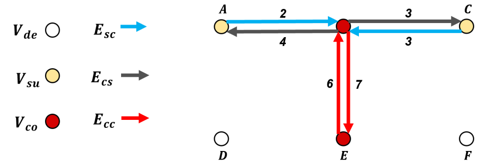


Fig. 3: HMOG Graph

between two vertices, from $u \in V_{co}$ to each $v \in M_k(u)$. Add these super-edges between core vertices to E_{cc}. (3) For $u \in V_{su}$, we find its $M_k(u)$ on $G - V_{de}$ and $M^r_k(u)$ on the reversed graph. The super-edges built to $M_k(u)$ are stored into edge set E_{sc}, each represents a path from a sub-level vertex to a core vertex. Similarly, the super-edges built to $M^r_k(u)$ are stored into edge set E_{cs} for core-to-sub-level vertex pairs. (4) In the middle of finding $M_k(u)$ for each $u \in V_{su}$, every $v \in V_{su}$ sharing a shortest path with u without core vertex generates a sub-to-sub level super-edge, added to E_{ss}.

We can calculate the exact k-shortest path using the three edge sets [33]. Consider the following cases: **(i)** $u, v \in V_{co}$. We simply return the query result on graph $G^* = (V_{co}, E_{cc})$; **(ii)** $u, v \in V_{sub}$. First we check the super-edges of u in E_{ss}. If u can reach v directly, return the weight of super-edge. Otherwise, follow the general cases in **(iii, iv)**; **(iii)** $u \in V_{co}$. We need an additional step to add u with its super-edges $(u, \cdot) \in E_{sc}$ into G^*; **(iv)** $v \in V_{co}$. Add v with its super-edges $(\cdot, v) \in E_{cs}$ into G^* and return the query result. Its correctness has been proved. For details, please refer to [33].

Our final output for HMPO graph $G_h(V, E_h)$ is consists of: V_{co}, which forms the graph G^* for query together with super-edges E_{cc}; V_{su}, of which each vertex is added to G^* with edge sets E_{sc}, E_{cs}, E_{ss} temporally for query; V_{de}, which is only called at the route planning stage with MC. $E_h = E_{cc} \cup E_{cs} \cup E_{sc} \cup E_{ss}$ and $V = V_{co} \cup V_{su} \cup V_{de}$.

Example 4. *Following the setting in Example 1, we construct the G_h in Figure. 3. $V_{de} = \{D, F\}$ are marked in white and their edges are removed. $V_{co} = \{B, E\}$ are marked in dark red. Sub-level vertices $\{A, C\}$ are marked in light yellow. Two core-to-core edges between them are added to E_{cc} and marked in red. Two edges from B to A and C are added to E_{cs} and marked in grey. Two edges from A and C to B are added to E_{sc} and marked in blue.*

VI. HMPO Graph Based Insertor

With the HMPO Graph, we introduce a new algorithm to boost the insertion phase for the solution of MORP problem.

Recall that with limited walking distance, the MP candidates $MC(u)$ for each $u \in V_p$ are close to each other. One interesting problem is, if we fail to insert a candidate $v \in MC(u)$, do the rest $MC(u) - \{v\}$ help? To answer it, we define a new distance correlation, which bounds the time saving of switching to any vertex in $MC(u) - \{v\}$. If deducting the saving still cannot meet the time limitation, we can safely prune the whole set. However, to derive it on the traditional graph, we need all the distances from $|V|$ sources even though vertices in $MC(u)$ are close to each other. Fortunately, we find that based on the k-skip structure in our hierarchical graph, an upper bound can be derived effectively. Most of the related works only use k-skip cover for faster query and vertex clustering [11]. We notice that k-skip cover "cuts off" the shortest paths using the core vertices, which can be used as anchors to compare the difference between the paths from the same source to a pair of vertices. To the best of our knowledge, this is the first paper to explore this property of k-skip cover graph for task assignment.

In the following subsections, we first extract the important distance correlations in subsection VI-A, then devise an effective algorithm *SMDB* for MORP problem in subsection VI-B.

A. Maximum Difference for Reaching Distances

We first define *maximum difference* $MD(\cdot,\cdot)$ between two vertices as a general distance correlation as follows:

Definition 8. (Maximum Difference) Given a graph $G(V,E)$, for each pair of vertices $v_1, v_2 \in V$, the maximum difference for v_1 and v_2 is: $MD(v_1,v_2) = \max_{v_3 \in V}(SP_c(v_3,v_1) - SP_c(v_3,v_2))$.

In general, starting from any source $v \in V$, the largest difference between distances to reach two given vertices is denoted as their MD. However, calculating each correlation requires 2 single-source shortest-path queries on the reversed graph, which costs $O(|E| + |V|\log|V|)$.

In our setting, all candidates of a vertex are within a walking distance d_m. Hereby, their driving distances between each other are relatively short. Deriving their inner MDs could be easier. However, each insertion computation compares vertices within a group of MP candidates for source or destination. It is more reasonable to use a group-based relationship for maximum difference rather than pairwise MD.

Then, we try to determine a *checker* $Ch(u)$ for each MP candidate set and derive *set maximum difference*(SMD) as an upper bound of $MD^*(Ch(u), v, MC(u))$s of all vertices $v \in MC(u)$, where $MD^*(u_1,u_2,V_M)$ is a modified MD, of which sources exclude sub-level vertices that have super-edges to $v \in V_M$, that is,

$$MD^*(u_1,u_2,V_M) = \max_{v:(v,u)\notin E_{sc}\cup E_{ss}, u\in V_M}(SP_c(v,u_1) - SP_c(v,u_2)),$$

where E_{sc} and E_{ss} are super-edge sets in Section V-C.

For each $u \in V$, once a checker $Ch(u) \in MC(u)$ is selected, a brute way to derive all pairs of $MD(Ch(u), v)$ for each $v \in MC(u)$ is to derive all the distances from vertices in V as ***sources*** to v by using single-source shortest-path query, which costs $O(nc_m(|E| + |V|\log|V|))$ in all. Precalculation for all the $u \in V$ has $O(nc_m|V|(|E| + |V|\log|V|))$ time complexity, which is not applicable. To deal with it, we devise an algorithm named HMPO Graph-based Maximum Difference Generator (HMDG), which not only efficiently finds the upper bounds $SMD(\cdot)$ according to $Ch(\cdot)$, but also finds the $Ch(\cdot)$ for each $MC(\cdot)$ to minimize $SMD(\cdot)$. Instead of searching with sources from V, our algorithm uses sources within a small set VC and maintains the correctness, which utilizes the attributes underlying our hierarchical structure.

We present the details to find $Ch(u)$ and $SMD(u)$ for each $u \in V$ in Algorithm. 4. To calculate the differences, we first find the sources and derive shortest distances. The checked sources VC are constructed as the union of all core vertices, which have super-edges pointing to $v \in MC(u)$. Vertices in $MC(u)$ are close to each other, so VC is a small set. Lines 3-10 derive all pairs of the shortest distances from vertices $vc \in VC$ to vertices $v \in MC(u)$. As $VC \subseteq V_{co}$, if $v \in V_{co}$, we

Algorithm 4: HMPO graph-based Max Difference Generator

Input: HMOG graph $G_h = (V, E_h)$ with reversed edges, MP candidate sets MC.
Output: Checker Ch and Set Max Diff SMD for each MP candidate set

1 **foreach** $u \in V$ **do**
2   Build set $VC = \{vc|(vc,v) \in E_{cs}, v \in MC(u)\}$
3   Initialize dictionary CC for costs from VC to $MC(u)$
4   **foreach** $v \in MC(u)$ **do**
5   **if** $v \in V_{co}$ **then**
6   Run Dijstra Algorithm from source v on reversed Graph $G_r^* = (V_{co}, E_{cc})$ until all vertices $vc \in VC$ are visited. Record these costs into $CC[vc][v]$
7   **if** $v \in V_{su}$ **then**
8   Add v with its edges $(\cdot, v) \in E_{cs}$ into reversed G_r^*
9   Run Dijkstra Algorithm from source v until all vertices $vc \in VC$ are visited. Record these costs into $CC[vc][v]$
10   Remove v with its edges $(\cdot, v) \in E_{cs}$ from G_r^*
11   Initialize $V2MD[v] = 0$ for all $v \in MC(u)$
12   **foreach** $vc \in VC$ **do**
13   Find the minimal in $CC[vc][\cdot]$, denote the key as v^-
14   **foreach** $v^+ \in MC(u)$ **do**
15   **if** $V2MD[v^+] < CC[vc][v^+] - CC[vc][v^-]$ **then**
16         $V2MD[v^+] = CC[vc][v^+] - CC[vc][v^-]$
17   Find the minimal value of $V2MD[v]$, return $Ch(u) = v$ and $SMD(u) = V2MD[v]$
18 **return** Ch, SMD

run Dijkstra algorithm on reversed $G^* = (V_{co}, E_{cc})$ to get all the costs from $vc \in VC$ to v. The costs are stored in dictionary $CC[vc][v]$; if $v \in V_{su}$, we need to add v with its core-sub edges $(v,\cdot) \in E_{cs}$ into G^* before path query.

Secondly, we initialize a dictionary $V2MD(v) \rightarrow SMD$ for checker selection, which stores the $SMD(u)$ of choosing $v \in MC(u)$ as checker. Note that starting from each source $vc \in VC$, the difference of distances between vc to v_1 and vc to v_2 can be derived by $CC[vc][v_1] - CC[vc][v_2]$. We will further show that its maximum difference is the same as the global maximum in Lemma. VI.1. To find the best checker, for each $vc \in VC$, we find the vertex v^- which has minimal $CC[vc][v^-]$ in line 13, that is, among vertices in $MC(u)$, vc is the closest destination for vc. If we use v^+ as checker, the SMD is the maximal of $CC[\cdot][v^+] - CC[\cdot][v^-]$, which should be minimized. Lines 14-16 enumerate vc and update the $V2MD[v^+]$ if a higher maximum difference is found, that is, $V2MD[v^+] < CC[vc][v^+] - CC[vc][v^-]$. Finally, $V2MD[\cdot]$ saves the required SMD for each checker. We find the minimal value of $V2MD$ with its key v and return $Ch(u) = v$ and $SMD = V2MD[v]$.

So our algorithm finds $Ch(u)$ with $SMD(u)$ such that: (i) for any source $vc \in VC$ from nearby core vertices, $\forall v \in MC(u)$, $SP_c(vc, Ch(u)) - SP_c(vc, v) \leq SMD(u)$; (ii) $Ch(u)$ is chosen among $MC(u)$ to minimize $SMD(u)$. The following lemma states that $SMD(u)$ is not only an upper bound if sources are in VC, but also an upper bound for $MD^*(Ch(u), \cdot, MC(u))$, that is, source vertices from V except close sub-level vertices.

Lemma VI.1. *Our algorithm finds the valid global SMD as set maximum difference for each vertex $u \in V$, that is, if a vertex $l_c \in V$ has no super-edge $(l_c, v) \in E_{ss} \cup E_{sc}$ towards any $v \in MC(u)$, then $SP_c(l_c, Ch(u)) - SP_c(l_c, v) \leq SMD(u)$.*

Proof. For details, please refer to Appendix F in [3]. □

Time Complexity. The time costs of Lines 2 and 3 are $O(nc_m)$. There are $O(nc_m)$ iterations in lines 4-10 and each iteration costs $O(\sigma^* \log(\sigma^*))$, where σ^* refers to the total edges covered by a subgraph in G_h, on which any shortest path is at most twice of the maximum driving distance between two vertices within walking distance $\leq 2d_m$. It is sensitive to k and d_m but would not become slower with larger $|V|$. Line 11 costs $O(nc_m)$ time. Line 13 costs nc_m to find the minimum value and lines 14-16 form $O(nc_m)$ iterations taking $O(1)$ time in each iteration. For the size of VC, we borrow the definition $\bar{\sigma}_k$ from [33], where $\bar{\sigma}_k$ is the average number of k-hop neighbors of the vertices in V. Thus, there are $O(|VC|) = O(\bar{\sigma}_k nc_m)$ iterations in lines 12-16. Their total time complexity is $O(\bar{\sigma}_k nc_m^2)$. Line 17 cost $O(nc_m)$ to find the minimum. So the total time complexity of the big loop in lines 1-17 is $O(|V| (nc_m \sigma^* \log(\sigma^*) + \bar{\sigma}_k nc_m^2))$, where both σ^* and $\bar{\sigma}_k$ depend on the structure of the road network instead of the size, the total time complexity grows linearly with $|V|$.

B. SMD-Boost Algorithm

We use SMD to boost insertion in this subsection.

Recall that whenever we try to insert a request r_j into a route, the rider should be picked up before tp_j. For each position to insert the pick-up, we first check whether using $Ch(s_j)$ as the MP can catch the deadline. If it is not insertable and misses its deadline within a "timeout", we need a substitution for it which arrives earlier. In the last subsection, we find the maximum time saving, SMD. If it is smaller than the minimum "timeout" we need, we cannot pick up the request using any MP in time.

With the checker and set maximum difference, we illustrate our algorithm *SMDBoost* for the insertion phase in Algorithm. 5. Note that we add one more set for pruning, dead vertices DV. It means that no driver w_i with current location $l_i \in DV$ can serve this rider. We initialize $DV = \emptyset$ for each new request. Assume that we try to insert rider r_j into the route of driver w_i. First, if $l_i \in DV$, we can prune driver w_i. Otherwise, we derive arriving time $arv[\cdot]$ for each route vertex according to [36]. In line 4, all the sub-level vertices which have super-edges towards $MC(s_j)$ are collected in Ne. The distances between these vertices and an MP can be arbitrarily short through super-edges in $E_{sc} \cup E_{ss}$.

Pruning strategy in lines 5-12 finds the largest index id^* to insert any pick-up in $MC(s_j)$. We initialize $id^* = |S_{w_i}|$. Then we check each vertex $v \in S_{w_i}$ in order. As the distances from vertices in Ne to some MPs in $MC(s_j)$ can be arbitrarily short, if $v \in Ne$, it is possible to insert rider after v. In this case, we continue to check the next insertion position in line 8. For each vertex $v \notin Ne$, it can be viewed as a source for $MC(u)$ subject to $SMD(u)$. So if $arv[v] + SP_h(v, Ch(s_j)) - SMD(Ch(s_j)) \geq tp_j$, according to Lemma.VI.1, inserting

---

Algorithm 5: SMDBoost

Input: a driver w_i with route S_{w_i}, request r_j, MP candidate set MC, set maximum difference SMD, checker set Ch, dead vertices DV

Output: a route S_w^* for the driver w and updated DV

1 **if** *Driver's location* $l_i \in DV$ **then**
2   Return S_{w_i} and DV without insertion
3 Generate arriving time $arv[\cdot]$ for S_{w_i}
4 Collect all sub-level vertices which have super-edges to vertices in $MC(s_j)$ into set Ne
5 The largest index to insert pick-up: $id^* = |S_{w_i}|$
6 **foreach** $v \in S_{w_i}$ **do**
7   **if** $v \in Ne$ **then**
8   Continue
9   **if** $arv[v] + SP_h(v, Ch(s_j)) - SMD(Ch(s_j)) \geq tp_j$ **then**
10   **if** v=l_i **then**
11   Add l_i to DV. Insertion fails and returns Null
12   Record $id^* = idx(v) - 1$
13   Break
14 Insert r_j with adapted insertion algorithm where insertion indexes of pick-ups larger than id^* are pruned.
15 **return** S_w^*, DV

---

any MP after v misses the deadline tp_j. Insertion position can only be smaller than index of v denoted as $idx(v)$, that is, $id^* = idx(v) - 1$. A special case is that if inserting after the driver's current location l_i cannot catch t_j, driver w_i and all the other drivers at l_c currently can not serve r_j. So we add l_i into DV for future pruning.

After the checking phase, we insert r_j without checking indexes after id^* for the pick-up point. The base algorithm adapts the linear insertion algorithm [36] for MPs.

Time Complexity. Line 3 is a linear operation according to [36]. Line 4 costs $O(nc_m)$. There is an $O(|S_{w_i}|)$ loop in lines 5-12. All other lines from 1 to 12 cost $O(1)$. As the algorithm in line 13 is linear, the total time complexity is $O(|S_{w_i}|)$.

Lemma VI.2. *Pruning Algorithm 5 has no performance loss.*

Proof. For details, please refer to Appendix G in [3]. □

VII. Experimental Study

A. Experimental Methodology

Data set. We use both real and synthetic data to test our HMOG Graph-Based solution. Specifically, for the real data, we use a public data set NYC [4]. It is collected from two types of taxis (yellow and green) in New York City, USA. We use all the request data on December 30th to simulate the ridesharing requests in our experiments. Each taxi request in NYC contains the latitudes/longitudes of its source/destination locations, its starting timestamp, and its capacity. We can generate a ridesharing request and initialize its locations, release time, and capacity correspondingly.

In addition, we derive the distribution of requests of all the NYC requests in December and generate 4 synthetic datasets (Syn) with request size 100k, 200k, 400k, and 800k. We download the road network of NYC from Geofabrik [2]. It includes the labels of roads for both driving and walking. We clean it into the directed car graph G_c and passenger graph G_p according to the labels. This road network has been widely used as a benchmark for ridesharing studies [36].

The settings of our dataset are summarized in Table IV.

TABLE IV: Setting of Dataset and Model

Parameters	Settings
Number of vertices of NYC	57030
Number of edges of NYC	122337
Number of valid requests of NYC	277410

Implementation. We follow the common settings for simulating ridesharing applications in [9], [23], [36]. While building the graph for road network, the weights of edges are set to their time cost (divide the road length on Geofabric by the velocity of its road type). For each request, we map its source and destination to the closest vertex in the road network. The initial location of each driver is randomly chosen.

The details of setting default values for major parameters is discussed in Appendix H in [3]. Table V summarizes them (default values are in bold font).

The experiments are conducted on a server with Intel(R) Xeon(R) E5 2.30GHz processors with hyper-threading enabled and 128 GB memory. The simulation implementation is single-threaded, and the total running time is limited to 14 hours for NYC. In reality, a real-time solution should stop before its time limit (24 hours for us) [23], [36]. All the algorithms are implemented in Java 11. We first preprocess our road network according to Section V. The modified vertices and weighted edges can be loaded directly for route planning. We boost the shortest distance and path queries with an LRU cache according to the setting of previous works [23], [36].

Compared Algorithms. We compare SMDB with the state-of-the-art algorithms for route planning of ride-sharing.

- **GreedyDP** [36]. It uses a greedy strategy for route planning without MPs. Each request is assigned to the feasible new route with minimum increased cost.
- **BasicMP**. It is an extension from GreedyDP by adapting MPs to solve the MORP problem.
- **HSRP**. It uses the HMPO Graph to improve the effectiveness of BasicMP without pruning.

Metrics. All the algorithms are evaluated in terms of total unified cost, served requests $|\hat{R}|$and response time (average waiting time to arrange a request, *resp. time* for short), which are widely used as metrics in large-scale online ride-sharing proposals [23], [26], [36].

B. Experimental Results

In this subsection, we present the experimental results.

Impact of Number of Drivers $|W|$. The first column of Figure 4 presents the results with different numbers of drivers in NYC. Compared with GreedyDP, BasicMP outperforms it in terms of the number of served requests by 6.6% to 12.7% with the help of MPs, while SMDB and HSRP outperforms it by 21.4% to 29.9%. More requests are served with more drivers, results in a decrease of unified costs and an increase in the served rates of all the algorithms. BasicMP decreases the cost by 2.7% to 13.2% and SMDB decreases it by 4.5% to 32.7%. GreedyDP runs the fastest without MPs. SMDB runs faster than HSRP and BasicMP with a *resp. time* $< 0.2s$.

Impact of Capacity of Drivers a_w. The second column of Figure 4 presents the effect of the capacities of drivers. BasicMP serves 8.6% to 12.2% more requests than GreedyDP with 5.1% to 9.2% less cost. SMDB outperforms other algorithms on both serving rate, 27.3% to 33.1% higher than GreedyDP, and unified cost, 13.0% to 20.5% less than GreedyDP. With a larger capacity, all the algorithms serve more requests with lower costs. However, the improvement with capacity from 4 to 10 is not as significant as from 2 to 4. The reason is that the extra spaces are mostly wasted. As for response time, GreedyDP still runs faster without MPs. SMDB costs less time than BasicMP and HSRP.

Impact of Deadline Coefficient e_r. The third column of Figure 4 shows the results of varying the deadline coefficient e_r. With larger e_r, all the algorithms serve more requests with a lower unified cost. SMDB still serves more requests with lower cost, which outperforms GreedyDP by serving 26.6% to 30.3% more requests than GreedyDP and decreasing its cost by 9.8% to 18.7%. With meeting points, BasicMP increases the number of served requests by 8.0% to 12.6% and decreases the cost by 3.4% to 9.2% compared with GreedyDP. With a larger deadline coefficient e_r, they find more feasible routes and serve more requests. Time cost increases with larger e_r as each request can find more feasible candidate routes. GreedyDP still runs fastest and SMDB is faster than BasicMP and HSRP.

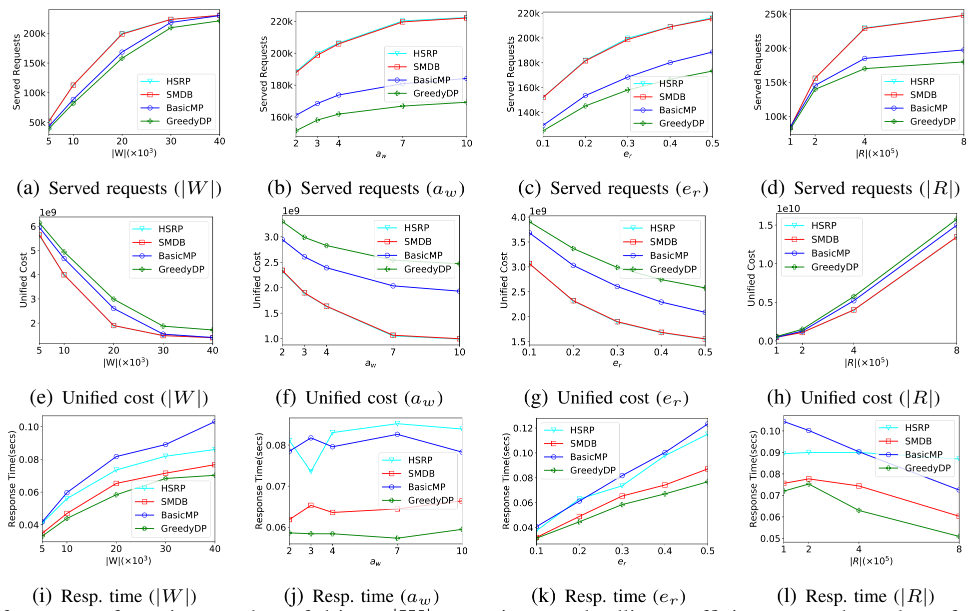
Impact of Number of Requests $|R|$. The fourth column of Figure 4 displays the results on different sizes of synthetic requests. The datasets are generated based on the distribution of all the NYC requests in December. All the algorithms serve more requests with a lower unified cost as the $|R|$ increases. Comparing with each other, SMDB serves 7.3% to 28.4% more requests than GreedyDP and decreasing its cost by 10.6% to 21.8%. BasicMP works weaker that increases the number of served requests by 32.5% to 10.8% and decreases the cost by 5.0% to 14.8% compared with GreedyDP. GreedyDP takes the shortest time followed by SMDB.

Summary of Results.

- Our SMDB algorithm can serve 7.3% to 33.1% more requests than the state-of-art algorithm [36]. The unified cost is decreased by 4.5% to 32.7%. These results validate the effectiveness of our algorithm in large scale datasets.
- With MPs, BasicMP, HSRP, and SMDB outperform GreedyDP. SMDB potentially arranges more requests on the highway and prunes candidates, which uses less time to serve more requests than BasicMP and HSRP. With response time lower than 0.2 seconds, SMDB is acceptable to be used as a real-time solution for ridesharing tasks.

VIII. Conclusion

In this paper, we propose the MORP problem, which utilizes meeting points for better ride-sharing route planning. We formulate a modified objective function to cover the cost of walking. We prove that the MORP problem is NP-hard and there is no polynomial-time algorithm with a constant competitive ratio for it. To cut the search space of meeting



(a) Served requests ($|W|$) (b) Served requests (a_w) (c) Served requests (e_r) (d) Served requests ($|R|$)

(e) Unified cost ($|W|$) (f) Unified cost (a_w) (g) Unified cost (e_r) (h) Unified cost ($|R|$)

(i) Resp. time ($|W|$) (j) Resp. time (a_w) (k) Resp. time (e_r) (l) Resp. time ($|R|$)

Fig. 4: Performance of varying number of drivers $|W|$, capacity a_w, deadline coefficient e_r, and number of requests $|R|$

points for fast assignemnts, we devise an algorithm to prepare meeting point candidates for each vertex. Besides, we construct an HMOG graph with hierarchical order on vertices for fast route planning, which takes the advantage of flexibility from meeting points to improve efficacy. Based on it, we propose SMDB algorithm to solve MORP problem effectively and efficiently. Extensive experiments on real and synthetic datasets show that our proposed solution outperforms baseline and the state-of-the-art algorithms for traditional ridesharing in effectiveness greatly without losing too much efficiency. Our paper is provided as a comprehensive theoretical reference for optimizing route planning with meeting points in ridesharing.

References

[1] [online] didi chuxing. http://www.didichuxing.com/.
[2] [online] geofabrik. https://download.geofabrik.de/.
[3] [online] technical report. https://cspcheng.github.io/pdf/MORP.pdf.
[4] [online] tlc trip record data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.
[5] [online] express pool. https://www.uber.com/us/en/ride/express-pool/, 2019.
[6] [online] uber express just like a bus. https://gizmodo.com/i-tried-uber-s-new-pool-express-service-and-honestly-j-1823190462, 2019.
[7] [online] open street map. https://www.openstreetmap.org/, 2020.
[8] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *PNAS*, 2017.
[9] M. Asghari, D. Deng, C. Shahabi, U. Demiryurek, and Y. Li. Price-aware real-time ride-sharing at scale: an auction-based approach. In *SIGSPATIAL*. ACM, 2016.
[10] M. Asghari and C. Shahabi. An on-line truthful and individually rational pricing mechanism for ride-sharing. In *SIGSPATIAL*. ACM, 2017.
[11] P. Cheng, H. Xin, and L. Chen. Utility-aware ridesharing on road networks. In *SIGMOD*. ACM, 2017.
[12] B. Cici, A. Markopoulou, and N. Laoutaris. Designing an on-line ride-sharing system. In *SIGSPATIAL*. ACM, 2015.
[13] P. Czioska, D. C. Mattfeld, and M. Sester. Gis-based identification and assessment of suitable meeting point locations for ride-sharing. *Transportation Research Procedia*, 2017.
[14] P. M. d'Orey, R. Fernandes, and M. Ferreira. Empirical evaluation of a dynamic and distributed taxi-sharing system. In *ITSC*. IEEE, 2012.
[15] E. Eser, J. Monteil, and A. Simonetto. On the tracking of dynamical optimal meeting points. *IFAC-PapersOnLine*, 2018.
[16] E. Feuerstein and L. Stougie. On-line single-server dial-a-ride problems. *Theor. Comput. Sci.*, 2001.
[17] S. Funke, A. Nusser, and S. Storandt. On k-path covers and their applications. *PVLDB*, 2014.
[18] R. Gandhi, S. Khuller, and A. Srinivasan. Approximation algorithms for partial covering problems. *J. Algorithms*, 2004.
[19] T. Gschwind and S. Irnich. Effective handling of dynamic time windows and its application to solving the dial-a-ride problem. *Transportation Science*, 2015.
[20] A. Gupta, M. T. Hajiaghayi, V. Nagarajan, and R. Ravi. Dial a ride from *k*-forest. *ACM Trans. Algorithms*, 2010.
[21] W. Herbawi and M. Weber. A genetic and insertion heuristic algorithm for solving the dynamic ridematching problem with time windows. In *GECCO*. ACM, 2012.
[22] S. C. Ho, W. Szeto, Y.-H. Kuo, J. M. Leung, M. Petering, and T. W. Tou. A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B: Methodological*, 2018.
[23] Y. Huang, F. Bastani, R. Jin, and X. S. Wang. Large scale real-time ridesharing with service guarantee on road networks. *PVLDB*, 2014.
[24] J.-J. Jaw. *Solving large-scale dial-a-ride vehicle routing and scheduling problems*. PhD thesis, Massachusetts Institute of Technology, 1984.
[25] A. Kleiner, B. Nebel, and V. A. Ziparo. A mechanism for dynamic ride sharing based on parallel auctions. In *IJCAI*, 2011.
[26] S. Ma, Y. Zheng, and O. Wolfson. T-share: A large-scale dynamic taxi ridesharing service. In *ICDE*. IEEE, 2013.
[27] S. Ma, Y. Zheng, and O. Wolfson. Real-time city-scale taxi ridesharing. *IEEE Trans. Knowl. Data Eng.*, 2015.
[28] M. Ota, H. T. Vo, C. T. Silva, and J. Freire. Stars: Simulating taxi ride sharing at scale. *IEEE Trans. Big Data*, 2017.
[29] Z. B. Rubinstein, S. F. Smith, and L. Barbulescu. Incremental management of oversubscribed vehicle schedules in dynamic dial-a-ride problems. In *AAAI*, 2012.
[30] D. O. Santos and E. C. Xavier. Dynamic taxi and ridesharing: A framework and heuristics for the optimization problem. In *IJCAI*, 2013.
[31] M. Stiglic, N. Agatz, M. Savelsbergh, and M. Gradisar. The benefits of meeting points in ride-sharing systems. *Transportation Research Part B: Methodological*, 2015.
[32] SUMC. What is shared-use mobility? https://goo.gl/3Jw6z7, 2018.
[33] Y. Tao, C. Sheng, and J. Pei. On k-skip shortest paths. In *SIGMOD*. ACM, 2011.
[34] R. S. Thangaraj, K. Mukherjee, G. Ravari, A. Metrewar, N. Annamaneni, and K. Chattopadhyay. Xhare-a-ride: A search optimized dynamic ride sharing system with approximation guarantee. In *ICDE*. IEEE, 2017.
[35] Y. Tong, L. Wang, Z. Zhou, B. Ding, L. Chen, J. Ye, and K. Xu. Flexible online task assignment in real-time spatial data. *PVLDB*, 2017.
[36] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu. A unified approach to route planning for shared mobility. *PVLDB*, 2018.
[37] N. H. Wilson, R. Weissberg, B. Higonnet, and J. Hauser. Advanced dial-a-ride algorithms. Technical report, 1975.
[38] N. H. M. Wilson, R. W. Weissberg, and J. Hauser. Advanced dial-a-ride algorithms research project. Technical report, 1976.
[39] A. C. Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *FOCS*. IEEE, 1977.
[40] S. Yeung, E. Miller, and S. Madria. A flexible real-time ridesharing system considering current road conditions. In *MDM*. IEEE, 2016.
[41] M. Zhao, J. Yin, S. An, J. Wang, and D. Feng. Ridesharing problem with flexible pickup and delivery locations for app-based transportation service: Mathematical modeling and decomposition methods. *Journal of Advanced Transportation*, 2018.
[42] L. Zheng, L. Chen, and J. Ye. Order dispatch in price-aware ridesharing. *PVLDB*, 2018.

APPENDIX

A. Proof of Lemma III.1

Lemma III.1. *The MORP problem is NP-hard.*

Proof. The basic route planning for ridesharing problems, which only takes the driving cost and rejection cost into account, is NP-hard [36]. A reduction from it to the MORP problem can be established by setting $\beta = \infty$ to ban walking. So MORP problem is NP-hard. $\square$

B. Proof of Lemma III.2

Lemma III.2. *There is no randomized or deterministic algorithm guaranteeing constant CP for the MORP problem.*

Proof. By proving no deterministic algorithm can generate constant expected value (e.g., ∞) with a distribution of the input including the destinations of the requests, the previous work [36] guarantees that no randomized algorithm has a constant CP using Yao's Principle [39], which is also applicable for our problem. Our problem is a variant of the basic problem as stated in the proof of Lemma III.1. Thus, no randomized or deterministic algorithm guarantees constant CP for MORP. $\square$

C. Proof of Lemma V.1

Lemma V.1. *Removing all vertices selected by the DVS algorithm from G_c with their edges leads to no detour cost.*

Proof. Recall that if a shortest path query $SP_c(v_1, v_2)$ finds a route contains a vertex u in the middle, we need to find an alternative path without u after removing u from graph G_c. If the new route has higher cost, removing u results in a detour cost. We define the new car graph without V_{de} as $G_{c'} = G_c - V_{de}$. Hereby the proof is equivalent to proof $\forall v_1, v_2 \notin V_{de}, SP_c(v_1, v_2) = SP_{c'}(v_1, v_2)$, where $SP_{c'}(v_1, v_2)$ is the shortest distance query on graph $G_{c'}$. We prove it by construction, that is, given any shortest path on G_c from v_1 to v_2, where $v_1, v_2 \notin V_{de}$, we show that there is a path from v_1 to v_2 on $G_{c'}$ with the same cost.

We use $\{u_1, u_2, \cdots, u_{|V_{de}|}\}$ to denote the removed defective vertices in order. When we remove a u_k, any length-2 shortest path (v_x, u_k, v_y) must have a same cost substitution $SCS_{x,y} =(v_x, v_{s_1}, v_{s_2}$
$\cdots, v_{s_p}, v_y)$, where $v_{s_i} \notin \{u_q | 0 < q <= k\}$, $i = 1, 2, \cdots, p$. It is satisfied according to the phase **(iii)** of Algorithm. 2.

Then, for any path on G_c from v_1 to v_2, we can iteratively find each u_k with the lowest index. Denote its previous and latter vertices as v_x, v_y, we substitute (v_x, u_k, v_y) with $SCS_{x,y}$. In each round, the lowest index of u_k in the new path is increasing. After at most $|V_{de}|$ rounds, the path has no vertex in V_{de}. As each substitution does not increase the cost, the final substitution is a valid path on $G_{c'}$ with cost equal to $SP_c(v_1, v_2)$. $\square$

D. Proof of Lemma V.2

Lemma V.2. *$\forall u \in V$ is accessible after removing vertices selected by the DVS algorithm from G_c with the help of meeting points.*

Proof. After popping each vertex u in phase 3 of the DVS algorithm, we ensure that $\exists v \in MC(u)$ is in V_{de} to guarantee that any vertex without available meeting points is still in graph G_c. Once a request starts from or aims at u, we can serve it by u itself. On the other hand, if a vertex is added to V_{de}, we mark its meeting point candidates so that we would not further remove these vertices from G_c. Request with u as origin or destination can be served via its meeting point candidates in G_c. $\square$

E. Proof of Lemma V.3

Lemma V.3. *Assume that we have N vertices in total, with M set as optimal solution for the attribute* (ii)*, the upper bound of the size of core vertex set is $\sigma(k) = \max(\frac{N}{k}log\frac{N}{k}, nc_m \cdot M)$.*

Proof. We prove it by dividing all the cases into two types:

(1) $\boldsymbol{nc_m \cdot M \geq \frac{N}{k}log\frac{N}{k}}$. Step (1) returns V'_{co} with size $|V'_{co}| \leq nc_m \cdot M$. So after step (2), if the size of remaining vertices is larger than $nc_m \cdot M$, there must be some vertices $u \in V - V_{de} - V'_{co}$ left.

As [33] shows that any subset of V with size at least $\frac{N}{k}log\frac{N}{k}$ is a k-skip cover of V, step (2) can still prune some vertices $u \in V - V_{de} - V'_{co}$ without violating attribute *(i)*, which is a contradiction. Thus, at most $nc_m \cdot M$ vertices are left after step (2). As the remaining vertices including all the V'_{co}, which is a valid cover, the size of the final output is no larger than $nc_m \cdot M$.

(2) $\boldsymbol{nc_m \cdot M < \frac{N}{k}log\frac{N}{k}}$. Step (1) returns V'_{co} with size $|V'_{co}| \leq nc_m \cdot M < \frac{N}{k}log\frac{N}{k}$. So after step (2), if the size of the remaining vertices is larger than $\frac{N}{k}log\frac{N}{k}$, there must be some vertices $u \in V - V_{de} - V'_{co}$ left. This also implies that step (2) can still prune some vertices $u \in V - V_{de} - V'_{co}$ without violating the k-skip cover, which results in a contradiction. So at most $\frac{N}{k}log\frac{N}{k}$ vertices are left after step (2).

The step (3) will maintain the valid cover and reduce the size. So the final output size is no larger than $\frac{N}{k}log\frac{N}{k}$. $\square$

F. Proof of Lemma VI.1

Lemma VI.1. *Our algorithm finds the valid global SMD as set maximum difference for each vertex $u \in V$, that is, if a vertex $l_c \in V$ has no super-edge $(l_c, v) \in E_{ss} \cup E_{sc}$ towards any $v \in MC(u)$, then $SP_c(l_c, Ch(u)) - SP_c(l_c, v) \leq SMD(u)$.*

Proof. We prove it by contradiction. Given a vertex $u \in V$ and its $MC(u)$ with outputs $Ch(u)$ and $SMD(u)$, assume that $\exists l_c \in V$ has no super-edges in $E_{ss} \cup E_{sc}$ to $MC(u)$, $\exists v \in MC(u)$ which satisfies $SP_c(l_c, Ch(u)) - SP_c(l_c, v) > SMD(u)$. We denote the last core vertex in the path from l_c to v as v_c. Here we have

$$\begin{aligned}
&SMD(u) < SP_c(l_c, Ch(u)) - SP_c(l_c, v)\\
\leq&(SP_c(l_c, v_c) + SP_c(v_c, Ch(u))) - (SP_c(l_c, v_c) + SP_c(v_c, v))\\
=&SP_c(v_c, Ch(u)) - SP_c(v_c, v)
\end{aligned}$$

As v_c is the last core vertex along the path, v_c must be a k-SKIP NEIGHBOR of $v \in MC(u)$. Thus, we have:

$$\begin{aligned}
&SP_c(v_c, Ch(u)) - SP_c(v_c, v)\\
=&CC[v_c][Ch(u)] - CC[v_c][v] \leq SMD(u)
\end{aligned}$$

Otherwise, $SMD(u) = V2MD[Ch(u)]$ would be set to $CC[v_c][Ch(u)] - CC[v_c][v]$. Contradiction. Thus, proved. $\square$

TABLE V: Parameter Settings.

Parameters	Settings
Deadline Coefficient e_r	0.1, 0.2, **0.3**, 0.4, 0.5
Capacity a_w	2, **3**, 4, 7, 10
Driving Distance Weight α	**1**
Walking Distance Weight β	0.5, **1**, 1.5, 2
Penalty p_o	**30**
Number of drivers $\|W\|$	3k, 5k, **10k**, 15k, 20k

G. Proof of Lemma VI.2

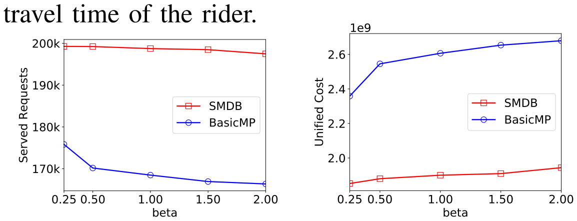
Lemma VI.2. *Pruning Algorithm 5 has no performance loss..*

Proof. Line 9 in Algorithm 5 guarantees that starting from v at time $arv[v]$ to pick up r_j directly misses the deadline tp_j. So inserting pick-up in latter indexes can be viewed as, starting from v to pick up r_j with some detour, which costs more. So the pruned insertion positions latter than v are not insertable in the original algorithm. Thus, our algorithm has no performance loss. □

H. Setting Default Values for Parameters

In real-application, the penalty of a rejection can be treated mainly as the money loss in proportion to the length of the tour (i.e. $p_j = p_o \times SP_c(s_j, e_j)$). The penalty weight p_o is usually greatly larger than the weight for travel cost α. This guarantees no request will be rejected if a feasible new route can serve it. Different p_o neither changes the assignment result nor affects the serving rate, so we do not need to compare it.

We set the delivery deadline of each request as the sum of its release time and the shortest time from source to destination extended by a Deadline Coefficient e_r. For example, the default deadline for a request with release time tr_j is $tr_j + (1+e_r) \cdot SP_c(s_j, e_j)$. We set the deadline for pick-up as the latest time, that is, tr_j minus the shortest time required to finish it after pick-up. a_j is varied from 2 to 10. The platform pays a driver for its travel time. We set the unit travel fee as the unit cost. (i.e. $\alpha = 1$). Walking distance results in a discount for riders. The unit walking time corresponds to β unit cost, which varies from 0.5 to 2. Its default value is set to 1, which ensures that using meeting points should not increase the travel time of the rider.



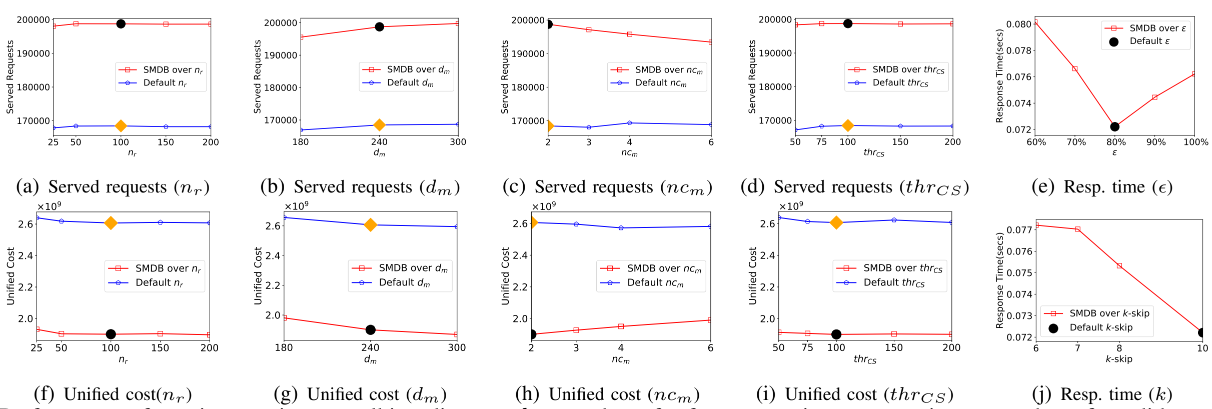
(a) Served requests (β) (b) Unified cost (β)

Fig. 5: Performance of varying β

Figure. 5 displays the effect of β. We propose $BasicMP$, which adapts the state-of-art traditional ridesharing solution to fit meeting points mode, as a baseline to compare with $SMDB$. We use the default setting in Table V to test them. Larger β leads to a larger cost of walking, thus decreases the flexibility. As β increases from 0.25 to 2.0, both of the two algorithms serve fewer requests and cost higher. However, $BasicMP$ works worse and serves 12.2% fewer requests while $SMDB$ serves 7.8% fewer. The reason is that $SMDB$ exploit the benefit of flexibility by arranging routes with convenient vertices. $BasicMP$ only cares about the temporal cost and loses a lot.

Besides, we conduct extensive experiments to compare the impact of different parameters for meeting point candidate selection and HMPO graph construction. Note that the construction is offline and its time cost would not affect the online assignment. Here we display the results of different factors for meeting point candidate selection: maximum walking distance $d_m = [180, 240, 300]$; number of reference vertices $n_r = [25, 50, 100, 150]$; maximum number of candidates $nc_m = [6, 7, 10]$; and the threshold $thr_{CS} = [0, 50, 100, 200]$. To construct the HMPO Graph, we compare $\epsilon = [40\%, 60\%, 80\%, 100\%]$ and $k = [5, 8, 10, 15]$ for the k-skip cover. The generated candidates and HMPO graphs are applied to our $SMDB$ algorithm with the default setting in Table V. We show their performances in Figure. 6.

According to the results, we choose the best setting: $n_r = 100$, $nc_m = 7$, $thr_{CS} = 100$, $\epsilon = 80\%$, and 10-skip cover. All the chosen settings results in the lowest unified cost and highest serving rate except $nc_m = 7$, which has lowest cost but sub-optimal serving rate (compared with $nc_m = 6$). As for the maximum walking distance, a larger d_m always has better performance (more flexible choices for meeting points) but impairs the users' experience (walking farther). To increase d_m from 240 to 300 (25%), the number of served requests only increase 0.5%. Here we choose $d_m = 240$ as a trade-off.



(a) Served requests (n_r) (b) Served requests (d_m) (c) Served requests (nc_m) (d) Served requests (thr_{CS}) (e) Resp. time (ϵ)

(f) Unified cost(n_r) (g) Unified cost (d_m) (h) Unified cost (nc_m) (i) Unified cost (thr_{CS}) (j) Resp. time (k)

Fig. 6: Performance of varying maximum walking distance d_m, number of reference vertices n_r, maximum number of candidates nc_m, the threshold thr_{CS}, ϵ, and k for the k-skip cover.