A Hierarchical Grouping Algorithm for the Multi-Vehicle Dial-a-Ride Problem

Kelin Luo
University of Bonn
Bonn, Germany
kluo@uni-bonn.de

Alexandre M. Florio
Polytechnique Montreal
Montreal, Canada
aflorio@gmail.com

Syamantak Das
Indraprastha Institute of Information Technology Delhi
Delhi, India
syamantak@iiitd.ac.in

Xiangyu Guo
University at Buffalo
Buffalo, USA
xiangyug@buffalo.edu

ABSTRACT

Ride-sharing is an essential aspect of modern urban mobility. In this paper, we consider a classical problem in ride-sharing – the Multi-Vehicle Dial-a-Ride Problem (Multi-Vehicle DaRP). Given a fleet of vehicles with a fixed capacity stationed at various locations and a set of ride requests specified by origins and destinations, the goal is to serve all requests such that no vehicle is assigned more passengers than its capacity at any point in its trip.

We give an algorithm HGR, which is the *first non-trivial approximation algorithm* for the Multi-Vehicle DaRP. The main technical contribution is to reduce Multi-Vehicle DaRP to a certain capacitated partitioning problem, which we solve using a novel hierarchical grouping algorithm.

Experimental results show that the vehicle routes produced by our algorithm not only exhibit less total travel distance compared to state-of-the-art baselines, but also enjoy a small in-transit latency, which crucially relates to each individual rider's traveling time. This suggests that HGR enhances rider experience while being energy-efficient.

PVLDB Reference Format:
Kelin Luo, Alexandre M. Florio, Syamantak Das, and Xiangyu Guo. A Hierarchical Grouping Algorithm for the Multi-Vehicle Dial-a-Ride Problem. PVLDB, 16(5): 1195 - 1207, 2023.
doi:10.14778/3579075.3579091

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/amflorio/hga-dial-a-ride.

1 INTRODUCTION

Over the last decade, ride-sharing has emerged as one of the most prominent aspects of shared economy [5]. In a typical ride-sharing scenario, riders with similar routes use a common vehicle for their commutes. The popularity of this framework has soared in recent years owing to the fact that all major urban taxi providers like Uber, Lyft and Didi Chuxing have introduced a 'carpooling' option. Economic benefits of ride-sharing are enjoyed by both the riders and providers: riders pay less for the same commute compared to hiring an individual taxi whereas the provider earns more profit in a single ride. Perhaps even more importantly, there is a potentially huge positive impact of ride-sharing on the environment [2]. Ride-sharing results in overall less fuel consumption and reduces air pollution by decreasing the number of vehicles on the road.

In order to reap the most benefit out of ride-sharing, it is essential to determine an efficient policy of assigning riders to vehicles. Owing to the large scale of the problem and the various constraints it might pose, there has been an increasing body of work in Computer Science and Operations Research that targets to design efficient algorithms to carry out such a task.

In this paper, we consider a classical problem in the area named the Dial-a-Ride Problem (DaRP) [9]. Informally, a mobility provider has a fleet of vehicles at their disposal, each with a certain capacity. There is a set of ride requests specified by origins and destinations. The algorithmic task is to assign every rider to exactly one vehicle and determine routes for the vehicles under the constraint that at any point during the trip, the vehicle must not accommodate more riders than its capacity. Finally, the goal is to minimize the total travel distance of all the vehicles. One can view this objective function as targeted towards lower total fuel consumption and pollution caused due to the commutes.

Heuristics for DaRP. Several heuristic approaches have been proposed for the DaRP over the years (see, for example, the survey [18]). In recent years, the size of ride-sharing problems to be solved has scaled almost exponentially, primarily owing to the advent of online platforms. This has attracted the attention of the database/datamining community to this problem and several techniques developed by these communities have been proven to be effective in handling ride haring requests at a massive scale [4, 19, 37, 38]. We highlight two recent algorithms which are state-of-the-art and have been experimentally established to be more effective than all the popular heuristics designed previously. The first one called pruneGDP was introduced by [37]. This is a fast algorithm that exploits a popular approach called insertion which has been utilized in solving dial-a-ride and its variants [4, 19–21, 28]. Roughly speaking, the algorithm maintains a partial assignment of requests (and hence routes) for each vehicle. At every iteration, the algorithm determines the assignment of one unassigned request to a

Proceedings of the VLDB Endowment, Vol. 16, No. 5 ISSN 2150-8097.
doi:10.14778/3579075.3579091

vehicle in a way that causes the minimal increment in total travel distance. The authors give an elegant $O(n)$-time implementation of this subroutine and experimentally demonstrate the effectiveness of this heuristic over several previous heuristics like [19, 28].

The second algorithm, FESI [38], is an approximation algorithm for the somewhat complementary objective of minimizing the makespan, that is, the maximum travel distance of any vehicle. In fact, the authors claim through empirical evidence that FESI is comparable to pruneGDP even for the total travel distance objective although it does not explicitly aim to minimize this.

Although these algorithms have been experimentally demonstrated to be effective and scalable, none of these works provide a formal worst case performance guarantee on the objective function value of total travel distance. In fact, for both algorithms, one can easily construct instances where their performance could be arbitrarily bad compared to an optimal solution.

Approximation Algorithms. There has been significant interest in the theoretical computer science community regarding DaRP. The problem is easily seen to be NP-hard even in the special case when every request has its origin and destination co-located – this is the classical Travelling Salesman Problem. For the special case of a *single vehicle* with capacity λ and n riders, two independent algorithms were given by Charikar and Raghavachari [3] and later on by Gupta et al. [17] with approximation guarantees of $O(\sqrt{\lambda}\log n)$ and $O(\sqrt{\lambda}\log^2 n)$, respectively. These are the best known theoretical guarantees so far. However, there is no approximation algorithm reported in the literature for the case of *multiple vehicle* DaRP that we consider.

The above discussion motivates the following question: *Is there an algorithm for multiple vehicle DaRP which is provably good compared to the optimal solution in the worst case ?* In this paper, we give the first non-trivial approximation algorithm for the multiple vehicle DaRP with an approximation ratio of $O(\sqrt{\lambda}\log n)$. Our approximation guarantee, perhaps surprisingly, does not depend on the number of vehicles and exactly matches the guarantee for the single vehicle case stated above. Our technique at a high level resembles the approach used in [17]. However, we need several non-trivial modifications and novel ideas to handle the multi-vehicle scenario. At the core, our algorithm uses a novel *hierarchical partitioning* of the rider set into groups which can be routed at a "small cost". These groups are then carefully assigned to vehicles followed by a routing phase for each vehicle. Whereas, for the single vehicle case, such a partitioning can be found by a relatively simple greedy approach, our algorithm needs to heavily utilize bipartite matching and ideas from routing literature which help us to bound from above the total travel distance of our algorithm. Our **key contributions** are as follows:

- We give the *first non-trivial approximation algorithm* for the multiple vehicle DaRP. We obtain an approximation factor of $O(\sqrt{\lambda}\log n)$, where λ is the capacity of the vehicles and n is the number of riders.
- Extensive experiments have been carried out to establish the practical efficacy of our algorithm. We compare our algorithm with state-of-the-art heuristics for DaRP. Our method outperforms all these algorithms on total travel distance by a significant margin of up to 30% on synthetic and real-world datasets.
- Our theoretical guarantees are valid only for the objective of minimizing the total travel time of the vehicles. However, in our experiments, we also consider the *in-transit latency of the riders*. This measures the amount of time a rider spends in the vehicle and can be thought of as a metric of rider experience. Empirical evidence shows that our proposed algorithm leads to an average in-transit latency up to 50% less than other DaRP algorithms.

2 RELATED WORK

Dial-a-Ride. The Dial-a-Ride problem has attracted significant attention, first from the operations research community, and later from the algorithms and database communities of computer science, owing to its vast application in real life. There have been attempts to design exact algorithms for the problem as early as 2006 by Cordeau [6], who proposed a branch-and-cut framework. This was followed up by several works that improved upon the basic framework by utilizing hybrid techniques like combining column-generation with dynamic programming [14]. However, as well known, such approaches search over a massive combinatorial space and do not scale. More practical approaches using Tabu Search [7], Simulated Annealing [29] and more efficient Neighborhood Searches [15] have also been considered. While promising to be more practical than the exact approaches, even the most efficient of these algorithm, ALNS [15], suffers at scale as shown in [38]. More recently, this problem has attracted the attention of the database/datamining community, particularly owing to the almost exponentially increasing demand of app-based ride-sharing. The pruneGDP algorithm of [37] remains the most efficiently and scalable state-of-the-art heuristic to the best of our knowledge.

None of the above approaches attempted to investigate practical algorithms with theoretical approximation guarantees. In fact, the literature there is surprisingly sparse. The only two algorithms providing approximation guarantees are given by Charikar and Raghavachari [3] and Gupta et al. [17], with approximation guarantees of $O(\sqrt{\lambda}\log n)$ and $O(\sqrt{\lambda}\log^2 n)$, respectively. We refer the interested reader to the excellent survey in [18] for a comprehensive landscape of Dial-a-Ride problems.

Other Ride-Sharing Problems. Several variants of ride-sharing problems have also been considered recently. For instance, the works of [37, 39–41] design heuristics to maximize revenue of the platform. The work in [38] designs approximation algorithm for the complementary objective of minimizing the maximum service time of individual requests. There have been interesting efforts to optimize more complex social utilities of both the platform and requests [4].

A slightly different, more static version of the DaRP has been considered in [1, 25, 27]. All these papers provide approximation algorithms to solve the problem where there are multiple vehicles with capacities and rider requests and the task is to make a static one-time assignment of the riders to the vehicles. However, none of these techniques directly work for the classical DaRP problem that we consider in this paper.

3 PRELIMINARIES

Problem definition. Let (V, d) be a given metric space, R be a set of n requests, where each request $r_i = (s_i, t_i) \in V \times V$ consists of a pickup location s_i and a drop-off location t_i. We also have a set of m vehicles K, where each vehicle $k \in K$ has a depot p_k and a capacity λ. Let V_K denote the multiset of all vehicle depot locations. The goal is to find an *assignment* $\mathcal{A}$ from vehicles to requests. An assignment is a collection of *walks*[1] in V, each of which starts from a distinct vehicle depot, and delivers a subset of requests from their pickup locations to drop-off locations.

Definition 1 (Vehicle Walk). *Given a set of requests R_k assigned to a vehicle $k \in K$, a **vehicle walk** is a sequence*

$$\mathsf{walk}_k = \langle \ell_0 = p_k, \ell_1, \ell_2, \cdots, \ell_t \rangle,$$

*starting at the origin location of vehicle k, where $\ell_i \in \{s_r : r \in R_k\} \cup \{t_r : r \in R_k\}, 1 \le i \le t$. A vehicle walk walk_k is **feasible** if **(i)** $\forall r \in R_k$, s_r appears before t_r in walk_k and **(ii)** at any time point of the vehicle walk the corresponding vehicle carries at most λ requests. Further, the **cost** of a walk walk_k is defined as $\mathsf{cost}(\mathsf{walk}_k) = \sum_{i=0}^{t-1} d(\ell_i, \ell_{i+1})$.*

A *feasible assignment* should deliver all requests, while ensuring that all vehicle walks are feasible. The objective is to minimize the total travel distance of all vehicle walks in a feasible assignment $\mathcal{A}$, denoted as $\mathsf{cost}(\mathcal{A})$, i.e. $\min \sum_{\mathsf{walk}_k \in \mathcal{A}} \mathsf{cost}(\mathsf{walk}_k)$.

Definition 2 (Multi-Vehicle DaRP). *Given a metric space (V, d), a set of n requests $R := \{s_i, t_i\}_{i=1}^{n} \in V^2$, and a set of m vehicles with locations $V_K := \{p_k\}_{k=1}^{m} \in V$ and a capacity λ, find a set of **minimum length vehicle walks** of the vehicles starting at $\{p_k\}_{k=1}^{m} \in V$ that moves each request r_i from its origin s_i to its destination t_i such that each vehicle carries at most λ requests at any point along the walk.*

We say that a request is preempted if, after being picked up from its origin, it is left temporarily at some vertex before being picked-up again and delivered to its destination. In our setting this is not allowed, as we study the **non-preemptive** DaRP. Finally, when referring to set of requests, we view it both as a set of pairs in $V \times V$ and as a subset of V, where in the latter case it contains all pickup and drop-off locations appearing in the requests. Which viewpoint is being used should be clear from the context.

Example 1. *We use Figure 1 as a running example. Given 2 vehicles originally located at p_1 and p_2, and 8 customer requests r_i ($i \in [8]$) where customer i aims to travel from s_i to t_i; The vehicle capacity is 4 and the distance metric is denoted as d. We would like to find two vehicle walks starting at $\{p_k\}_{k=1}^{2}$ that moves each r_i from its origin s_i to its destination t_i such that each vehicle carries at most 4 requests at any point along the walk. There are 2^8 possible ways to assign the 8 requests to the two vehicles, and a vehicle can have many different orders to serve the assigned requests. For example, if r_1, r_2 are assigned to vehicle 1, then there are 6 feasible walks:*

$$\langle p_1, s_1, t_1, s_2, t_2 \rangle, \langle p_1, s_1, s_2, t_1, t_2 \rangle, \langle p_1, s_1, s_2, t_2, t_1 \rangle,$$

$$\langle p_1, s_2, t_2, s_1, t_1 \rangle, \langle p_1, s_2, s_1, t_2, t_1 \rangle, \langle p_1, s_2, s_1, t_1, t_2 \rangle.$$

The cost of serving requests is equal to the total length of the vehicle walks. For example, the cost of a walk $\langle p_1, s_1, t_1, s_2, t_2 \rangle$ is equal to $d(p_1, s_1) + d(s_1, t_1) + d(t_1, s_2) + d(s_2, t_2)$. Then, the optimal solution is the minimum length vehicle walks that serve all requests.

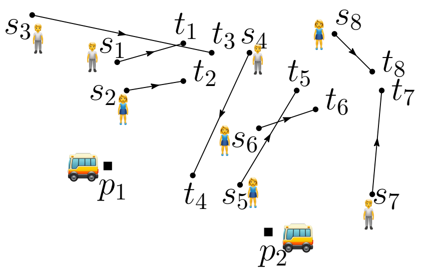


Figure 1: DaRP example

4 HGR: A NEW ALGORITHM FOR DARP

In this section, we introduce a novel $O(\sqrt{\lambda} \log n)$-approximation algorithm, which we call <u>H</u>ierarchical <u>G</u>rouping and <u>R</u>outing (HGR), for the Multi-Vehicle DaRP.

Before going in to the details of the algorithm and proof of approximation ratio, we give some high level ideas about our main techniques. The first idea towards designing this algorithm is to partition the requests into disjoint *groups* of size at most λ. The intent is to let a vehicle start empty, and serve one group entirely before moving on to the next one. Note that an optimal solution to this problem does not necessarily follow this strategy. However, the authors in [17] shows that there always exists a *near-optimal* solution which follows such a strategy. The following fact makes this formal.

Fact 1 (Solution Structure). [17] *Given any DaRP instance, there exists a feasible walk τ satisfying the following conditions:*

- *τ can be split into a set of segments $\{S_1, ..., S_t\}$ where each segment S_i services a set O_i of at most λ requests such that S_i is a path that first picks up each request in $O_i \subseteq R$ and then drops each of them.*
- *The length of τ is at most $O(\log n)$ times the length of an optimal walk.*

Although the above fact has been only proven for the single-vehicle DaRP in [17], it is not difficult to generalize this to the case of Multi-Vehicle DaRP: the fact directly implies that for each of the optimal vehicle walk in the Multi-Vehicle DaRP, there exists a feasible walk τ with a length no more than $O(\log n)$ times the optimal walk length; Since the length of the Multi-Vehicle DaRP is the sum of the all vehicles walk lengths, Fact 1 still holds for the Multi-Vehicle DaRP. The authors in [17] exploit this fact to design a *greedy* algorithm for the single vehicle case which works roughly as follows. The algorithm is iterative, where at each iteration, a group of λ requests (except possibly at the last iteration which might have less than λ requests) are formed. The criteria to form a new group is that among the remaining requests, they can be served by travelling the *minimum total distance.* A significant challenge in [17] was to design a method to make the greedy choice in each iteration. This

[1]A walk is a finite-length sequence of vertices $v_1, v_2, ..., v_N \in V$ for some N, and the cost (length) of the walk is defined as $\sum_{i=1}^{N-1} d(v_i, v_{i+1})$.

requires them to solve a highly non-trivial problem they call λ-*forest*, which is closely related to a notoriously hard problem called the λ-densest sub-graph problem. Indeed, the main contribution of the above paper was giving a $O(\sqrt{\lambda})$-approximation algorithm for λ-forest. This was combined with a standard argument from approximation algorithms literature to show that the overall approximation guarantee is $O(\lambda \log^2 n)$. However, this approach poses the following challenges in being effective as a practical algorithm for the multiple-vehicle case.

(1) The algorithm they use to solve λ-forest, although giving reasonable approximation guarantees, is highly complicated and not practical. In fact, we had implemented this approach for a single vehicle and found the running time scaling prohibitively with the number of requests (For 200 requests and capacity 16, it takes 1900 seconds, and for 500 requests and capacity 4, it takes 1500 seconds. We contrast this with our algorithm can handle say 10k requests with capacity 32 in about 100 seconds).
(2) It is not immediately clear how to adopt the above approach to the multiple vehicle case. Although Fact 1 still continues to be true, a major challenge here is to determine which group to assign to which vehicle (note that this issue does not exist in the single vehicle case). So, any algorithm which aims to provide a theoretical guarantee needs to combine the grouping phase and the assignment phase while not incurring a lot of cost.

In order to overcome these two issues, we avoid the *local greedy* approach of [17] and develop a novel algorithm that exploits a more *global* viewpoint. Our *core technical contribution* is to design a *hierarchical grouping technique* which avoids solving complicated problems like λ-forest. Instead we use relatively simpler sub-routines like bipartite matching and minimum spanning trees and still manage to obtain the same approximation ratio as [17] for the multiple vehicle case. This not only makes our algorithm more efficient and relatively easier to implement, we are also able to avoid the additional $\log n$ factor incurred by [17] due to the iterative greedy approach. Now we present the technical details of our algorithm.

As mentioned above, our main idea is to develop a hierarchical clustering algorithm to solve the grouping problem. Indeed, we partition requests into groups of size $\leq \lambda$ by considering the following capacitated grouping problem (see Definition 3), and give an approximation guarantee of $O(\sqrt{\lambda})$ (see Theorem 2 in Section 5).

Definition 3 (Capacitated Grouping Problem)**.** *Given an n-vertex metric space (V, d) and requests $R := \{s_i, t_i\}_{i=1}^m \in V^2$, find a set of* ***minimum length walks*** *that serves all requests and such that each walk covers at most λ requests.*

When talking about a walk in the capacitated grouping problem, it is always associated with the request group covered by it. A *feasible partition* $\mathcal{P}$ of R partitions R into groups of size at most λ. A **walk** covering request group $P \in \mathcal{P}$ is a sequence $\mathbf{w}_P = \langle \ell_1, \ell_2, \cdots, \ell_h \rangle$ that traverses P, where $\ell_i \in \{s_r : r \in P\} \cup \{t_r : r \in P\}, 1 \leq i \leq h$. A walk $\mathbf{w}_P$ is *feasible* if $\forall r \in P$, s_r appears before t_r in $\mathbf{w}_P$. The cost of $\mathbf{w}_P$ is denoted as $\text{cost}(\mathbf{w}_P) = \sum_{i=1}^{h-1} d(\ell_i, \ell_{i+1})$. The cost of partition $\mathcal{P}$ is denoted as $\text{cost}(\mathcal{P}) = \sum_{P \in \mathcal{P}} \min_{\text{feasible } \mathbf{w}_P} \text{cost}(\mathbf{w}_P)$.

Our main algorithm (Algorithm 1) for DaRP first treats the input as an instance of Capacitated Grouping Problem and solves it using Algorithm 2 to get a partition, then builds an actual route based on the partition. We now describe the main ideas in each step (See Figure 2 for an example).

Algorithm 1 HIERARCHICAL GROUPING AND ROUTING (HGR)

```
Input: Request set $R$, Vehicle locations $V_K$ and capacity $\lambda$
Output: A feasible assignment $\mathcal{A}$
1: $\mathcal{P} \leftarrow$ HIERARCHICAL GROUPING$(R, \lambda)$  // Alg. 2
2: $\mathcal{A} \leftarrow$ ROUTING$(R, \mathcal{P}, V_K)$  // Alg. 5
3: return $\mathcal{A} = \{\text{walk}_k : k \in K\}$
```

In Step 1, the HIERARCHICAL GROUPING algorithm partitions requests into groups such that the total length of walks covering the partition is not too large compared with the optimal solution. To achieve this goal, we develop a non-trivial two-layer hierarchical grouping technique: in the outer layer, iteratively combine two clusters of requests into one cluster; inside each cluster, we form groups to ensure that closer requests are grouped together and far-apart requests are divided into separate groups.

In Step 2, we use the partition obtained in Step 1 to design actual routes for all vehicles. The idea is to view each group as a single vertex, and compute a cheap spanning forest to assign vehicles to groups. The forest is computed such that each tree in it is rooted as some vehicle location of V_K, and this vehicle will traverse the tree to serve its requests in a group-by-group manner. We obtain the following result.

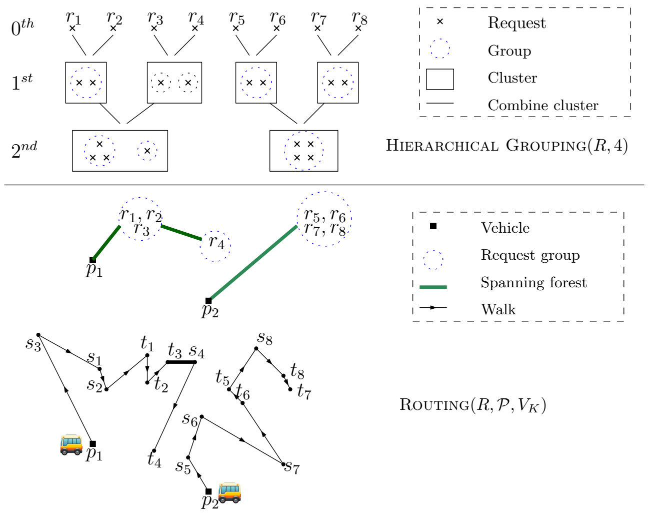
THEOREM 1. *Given a Multi-Vehicle DaRP with set of requests R and set of vehicles K, each with a capacity λ, the HGR algorithm runs in time $O(|R|^3 \log \lambda + |R|^2 \lambda^2 \log \lambda)$ and returns a set of $|K|$ feasible walks serving all requests in R such that the total travel distance is at most $O(\sqrt{\lambda} \cdot \log |R|)$ times that of an optimal solution to the Muil-Vehicle DaRP.*

Example 2. *Figure 2 shows running HGR on the Multi-DaRP instance given in Example 1 (Figure 1). HGR first invokes HIERARCHICAL GROUPING (Alg. 2) to partition requests into groups. In the outer layer of HG, we form clusters hierarchically. Here, there are 4 clusters in the 1st iteration: $\{\{r_1, r_2\}\}$, $\{\{r_3\}, \{r_4\}\}$, $\{\{r_5, r_6\}\}$ and $\{\{r_7, r_8\}\}$, and 2 clusters in the 2nd iteration: $\{\{r_1, r_2, r_3\}, \{r_4\}\}$, $\{\{r_5, r_6, r_7, r_8\}\}$. In the inner layer, we form groups inside each cluster based on the requests.*

- *If the requests of two clusters in the previous iteration are far-apart, although they are combined to one cluster, they are kept in separate groups. This is the case for the two clusters $\{\{r_3\}\}, \{\{r_4\}\}$ from the 0th iteration, which are combined into cluster $\{\{r_3\}, \{r_4\}\}$ in the 1st iteration.*
- *If some requests of two clusters in the previous iteration are close and they are combined to one cluster, then they are grouped together. This is the case for the two clusters $\{\{r_5, r_6\}\}, \{\{r_7, r_8\}\}$ from the 1st iteration, which are combined into cluster $\{\{r_5, r_6, r_7, r_8\}\}$ in the 2nd iteration in Figure 2.*

After $\log 4 = 2$ *iterations, we obtain the partition of the requests:* $\{r_1, r_2, r_3\}, \{r_4\}, \{r_5, r_6, r_7, r_8\}$.

In the second step, HGR invokes Routing *(Alg. 5) to build the final route based on the obtained partition. Each request group (with no more than λ requests) is viewed as a* point*, and we compute a minimum spanning forest over* 5 points $\{r_1, r_2, r_3\}, \{r_4\}, \{r_5, r_6, r_7, r_8\}$, p_1, p_2 *such that each tree is rooted at p_1, p_2. Then by traversing the tree to serve requests in a group-by-group manner, we obtain the two walks $\langle p_1, s_3, s_1, s_2, t_1, t_2, t_3, s_4, t_4\rangle$ and $\langle p_2, s_5, s_6, s_7, t_6, t_5, s_8, t_8, t_7\rangle$.*



0th
1st
2nd
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8
Request
Group
Cluster
Combine cluster
Hierarchical Grouping$(R, 4)$
r_1, r_2 r_3
r_4
r_5, r_6 r_7, r_8
p_1
p_2
Vehicle
Request group
Spanning forest
Walk
s_3 s_1 s_2 t_1 t_2 t_3 s_4 t_4
s_8 t_8 t_7 t_5 t_6 s_6 s_5 s_7
p_1 p_2
Routing$(R, \mathcal{P}, V_K)$

Figure 2: HGR for the DaRP example

We describe the HGR algorithm in more detail in Section 5 and Section 6. Proofs omitted due to space restrictions can be found in the full version [26].

5 PART (I): GROUPING

In this section, we present the Hierarchical Grouping (HG) algorithm, and use it to give an approximation guarantee of $O(\sqrt{\lambda})$ for the capacitated grouping problem. The intuition behind HG is to cluster requests iteratively such that in each iteration, the number of requests clustered together is doubled, so that after $\log\lambda$ iterations, every cluster contains λ requests. However, treating each cluster as a single group does not necessarily give small cost (i.e., travel length): In fact, this can be far from optimal, where good solutions may all use much smaller groups. Therefore, we develop the two-layer hierarchical grouping technique: In the outer layer, we form clusters hierarchically such that by combining 2 clusters of the $(i-1)$-th iteration, the number of requests in each cluster is 2^i in the i-th iteration, until each cluster contains λ requests[2]; In the inner layer, we form groups inside each cluster to ensure that closer requests are grouped together and far-apart requests are kept in separate groups. Notice that initially each cluster contains exactly one request. Since each iteration doubles the number of requests in each cluster and at the end each cluster contains no more than λ requests, the hierarchical grouping process will continue for $\lfloor\log\lambda\rfloor$ iterations.

[2]It also works for λ which is (possibly) not exact powers of 2: for an arbitrary λ with $2^i < \lambda < 2^{i+1}$, the process ends when each cluster contains 2^i requests. It does not affect the analysis.

The key idea is to define a suitable edge cost on a graph whose vertices represent the clusters formed in each iteration, and each iteration is carried out by computing a minimum-*cost* matching on this graph. The design of the edge cost function of the graph needs to be very careful, such that we can bound the cost of covering the groups with respect to the optimal cost.

Before formally presenting the algorithm, we define the following notations:

- (Request-) group $P, X \subseteq R$: a set of requests
- (Group-) cluster $\mathcal{P}, \mathcal{Q}, \mathcal{X}$: a set of groups
- (Cluster-) collection $\mathfrak{M}$: a set of clusters

Specifically, we use notation $w(\mathcal{X}, \mathcal{X}')$ to represent **cost function** on the edges (connecting two clusters $\mathcal{X}$ and $\mathcal{X}'$) of a graph. We first define the following notations for (request-) groups $X, X' \subseteq R$:

(1) Minimum Spanning Tree (MST) cost $\mathsf{mst}_{s,t}(X)$: $\mathsf{mst}_s(X)$ (resp. $\mathsf{mst}_t(X)$) is defined to be the cost of a MST over the origins (resp. destinations) of all requests in X. Furthermore, we define $\mathsf{mst}_{s,t}(X) = \mathsf{mst}_s(X) + \mathsf{mst}_t(X)$.
(2) (Incremental) Cost of serving groups together:
$$w_1(X, X') = \mathsf{mst}_{s,t}(X \cup X') - \mathsf{mst}_{s,t}(X) - \mathsf{mst}_{s,t}(X').$$
(3) Cost of serving groups separately:
$$w_2(X, X') = \min_{r_i \in X} d(s_i, t_i) + \min_{r_i \in X'} d(s_i, t_i).$$

During the execution of HG, we build a larger cluster by merging smaller clusters, and we can choose to either merge the groups *within* the smaller clusters or leave them separated. To guide the choice, we define the **cost function** of merging two clusters as follows: for every group pair $X \in \mathcal{X}, X' \in \mathcal{X}'$, we compare the (incremental) cost $w_1(X, X')$ of serving two groups together and the cost $w_2(X, X')$ of serving groups separately; then, we take the minimum as the cost of combining the two clusters (See Definition 4).

Definition 4 (Cost Function). *Given two clusters $\mathcal{X}$ and $\mathcal{X}'$, define*
$$w(\mathcal{X}, \mathcal{X}') = \min_{X \in \mathcal{X}, X' \in \mathcal{X}'} \min\{w_1(X, X'), w_2(X, X')\}$$
where w_1 and w_2 are defined as above.

We now describe the algorithm formally. In order to simplify the description, we assume for now that λ is a power of 2. It is straightforward to adapt the procedures for arbitrary λ with the same approximation guarantee.

In Algorithm 2, we iteratively build larger clusters in a hierarchical way, until every cluster is of size λ. Note that each cluster may still contain multiple request groups. We first initialize a trivial collection $\mathfrak{M}_0$ containing $|R|$ clusters, where each cluster contains one group and each group contains a distinct request (Line 1). Then, we repeat for $\log\lambda$ iterations, where in each iteration we compute a minimum weight perfect matching [3] on the current cluster collection (Line 7-8), and merge the matched clusters to form a collection for the next iteration (Line 9-13). Therefore, after each iteration, the number of clusters is halved, while the number of requests in each cluster is doubled. The weight function w defined in Definition 4

[3]The perfect matching is a classical problem in combinatorial optimization. The famous Edmond's algorithm [12, 36] solves the problem in polynomial-time and is usually invoked as a blackbox. In our computational experiments, we use a modern implementation of Edmonds's algorithm, available at [23].

is crucial for guiding the merge step (Line 9-13). When merging two clusters, we may also merge two groups from them (Line 11), or leave the groups untouched (Line 13), depending on whether w achieves its value via w_1 or w_2. Finally, we "unbox" the clusters and return all the groups formed.

Algorithm 2 HIERARCHICAL GROUPING(R,λ)

```
Input: $R$ and $\lambda$
Output: a partition $\mathcal{P}$, each group $P \in \mathcal{P}$ contains not more than $\lambda$ requests
1:  $\mathfrak{M}_0 = \bigcup_{r_i \in R}\{\{\{r_i\}\}\}$
2:  $\ell = 0$
3:  while $\ell < \log\lambda$ do
4:      $\ell = \ell + 1$
5:      $\mathfrak{M}_\ell = \emptyset$
6:      $\mathcal{P}_\ell = \emptyset$
7:      Let $G_\ell \equiv (\mathfrak{M}_{\ell-1}, e)$ be a complete graph with edge weights $w(Q, Q')$ for any $Q, Q' \in \mathfrak{M}_{\ell-1}$
8:      Find a minimum weight matching $M_\ell$ in $G_\ell \equiv (\mathfrak{M}_{\ell-1}, e)$ with total weight $w(M_\ell) = \sum_{(Q,Q')\in M_\ell} w(Q, Q')$
9:      for $(Q, Q') \in M_\ell$ do
10:         if $w(Q, Q') = w_1(P, P'), P \in Q, P' \in Q'$ then
11:             add cluster $Q \cup Q' \cup \{P \cup P'\} \setminus \{P, P'\}$ to collection $\mathfrak{M}_i$
12:         else if $w(Q, Q') = w_2(P, P'), P \in Q, P' \in Q'$ then
13:             add cluster $Q \cup Q'$ to collection $\mathfrak{M}_i$
14: $\mathcal{P}_l \leftarrow \{Q \in Q : Q \in \mathfrak{M}_l\}$ for $l = 1, \ldots, \log\lambda$
15: return $w(M_\ell)$ for $1 \le \ell \le \log\lambda$ and $\mathcal{P} = \mathcal{P}_{\log\lambda}$
```

We first prove some important properties of the HIERARCHICAL GROUPING (HG) algorithm that will help us to bound total cost of the capacity-bounded groups. Let $\mathcal{P}_l$ denote the l-th partition obtained by the HG Algorithm 2 and let $\mathcal{P}_{l,=i} \subseteq \mathcal{P}_l$ denote the request groups of size 2^i, i.e., $\mathcal{P}_{l,=i} = \{P \in \mathcal{P}_l : |P| = 2^i\}$, and let $\mathcal{P}_{l,<i}$ (resp. $\mathcal{P}_{l,>i}$) denote the request groups of size smaller than (resp. more than) 2^i, i.e, $\mathcal{P}_{l,<i} = \{P \in \mathcal{P}_l : |P| < 2^i\}, \mathcal{P}_{l,>i} = \{P \in \mathcal{P}_l : |P| > 2^i\}$. Based on the definition of the cost function w, we have the following lemma by a telescope sum:

Lemma 1. *For partition $\mathcal{P}_l$, $l \in [\log\lambda]$, obtained by Algorithm 2, we have*

$$\sum_{P\in\mathcal{P}_l}\left(\mathrm{mst}_{s,t}(P) + \min_{r_i\in P} d(s_i,t_i)\right) \le \sum_{i=1}^{l} w(M_i) + \sum_{P\in\mathcal{P}_{l,=l}} \min_{r_i\in P} d(s_i,t_i)$$

Due to space limit, we put the proof of Lemma 1 in the full version [26].

Next, we will bound the separate serving cost $\sum_{P\in\mathcal{P}} \min_{r_i\in P} d(s_i,t_i)$ (see Lemma 2) and $\sum_i w(M_i)$, respectively. We further introduce the following notations. Fix an optimal partition $\mathcal{P}^*$.

Lemma 2 (The Separate Serving Cost). *The costs of serving a request separately in each group of $\mathcal{P}_{l,=\log\lambda}$*

$$\sum_{P\in\mathcal{P}_{l,=\log\lambda}} \min_{r_i\in P} d(s_i,t_i) \le \sum_{P\in\mathcal{P}^\star} \max_{r_i\in P} d(s_i,t_i).$$

PROOF. Recall $\mathcal{P}_{l,=\log\lambda} = \bigcup_{P\in\mathcal{P},|P|=\lambda} P$. Consider a bipartite graph $(\mathcal{P}_{l,=\log\lambda}, \mathcal{P}^\star, E)$ where for each request $r \in \bigcup_{P\in\mathcal{P}_{l,=\log\lambda}} P$, there is an edge $e(r_i)$ between vertex $P \in \mathcal{P}_{l,=\log\lambda}$ and $\mathcal{P}^\star$. According to Hall's marriage theorem, we can find a matching M among $\bigcup_{P\in\mathcal{P}_{l,=\log\lambda}} P$ and $\mathcal{P}^\star$ that covers all $\mathcal{P}_{l,=\log\lambda}$. Let $E(M)$ be the edges in M. We have

$$\sum_{P\in\mathcal{P}_{l,=\log\lambda}} \min_{r_i\in P} d(s_i,t_i) \le \sum_{e(r_i)\in E(M)} d(s_i,t_i) \le \sum_{P\in\mathcal{P}^\star} \max_{r_i\in P} d(s_i,t_i).$$

□

Lemma 3 (The ℓ-th Grouping Cost). *The grouping cost in ℓ-th iteration is*

$$w(M_l) \le \sum_{P\in\mathcal{P}^\star} 2^{\frac{\log\lambda - l}{2}} \mathrm{mst}_{s,t}(P) + \sum_{P\in\mathcal{P}^\star} \max_{r_i\in P} d(s_i,t_i)$$

if $\log\lambda - l$ is even; otherwise,

$$w(M_l) \le \frac{3}{2} \cdot \sum_{P\in\mathcal{P}^\star} 2^{\frac{\log\lambda - l - 1}{2}} \mathrm{mst}_{s,t}(P) + \sum_{P\in\mathcal{P}^\star} \max_{r_i\in P} d(s_i,t_i).$$

Due to space limit, we leave the proof of this lemma in the full version of this paper [26]. Based on Lemma 3, it is easy to obtain the following lemma:

Lemma 4 (Total Grouping Cost). *The total grouping cost*

$$\sum_{l=1}^{\log\lambda} w(M_l) \le O(\sqrt{\lambda}) \sum_{P\in\mathcal{P}^\star} \mathrm{mst}_{s,t}(P) + \log\lambda \cdot \sum_{P\in\mathcal{P}^\star} \max_{r_i\in P} d(s_i,t_i).$$

Now we are ready to bound the cost of the capacity-bounded groups:

THEOREM 2. *The HIERARCHICAL GROUPING algorithm runs in polynomial time and outputs a feasible partition $\mathcal{P}$ of R, such that (1) $\forall P \in \mathcal{P}, |P| \le \lambda$; (2) $\mathrm{cost}(\mathcal{P}) \le O(\sqrt{\lambda}) \cdot \mathrm{cost}(\mathcal{P}^*)$, where $\mathcal{P}^*$ is the partition of an optimal solution for the capacitated grouping problem; and (3) $\forall P \in \mathcal{P}$ one can efficiently find a feasible walk $\mathbf{w}_P$ traversing P, such that $\sum_{P\in\mathcal{P}} \mathrm{cost}(\mathbf{w}_P) \le O(\mathrm{cost}(\mathcal{P}))$.*

PROOF. First notice that, given any group of request P, we have the following lowerbounds for the cost of the optimal walk

$$\mathbf{w}_P^* := \underset{\text{feasible } \mathbf{w}_P}{\arg\min}\ \mathrm{cost}(\mathbf{w}_P) : \mathrm{mst}_{s,t}(P) \le 2\mathrm{cost}(\mathbf{w}^*),$$

$$\min_{r_i\in P} d(s_i,t_i) \le \max_{r_i\in P} d(s_i,t_i) \le \mathrm{cost}(\mathbf{w}_P^*).$$

Then by Lemma 1, Lemma 2, and Lemma 4, we have

$$\begin{aligned}&\sum_{P\in\mathcal{P}} (\mathrm{mst}_{s,t}(P) + \min_{r_i\in P} d(s_i,t_i))\\ \le& O(\sqrt{\lambda}) \cdot \sum_{P\in\mathcal{P}^\star} \mathrm{mst}_{s,t}(P) + (1+\log\lambda) \cdot \max_{r_i\in P} d(s_i,t_i)\\ \le& O(\sqrt{\lambda}) \cdot \sum_{P\in\mathcal{P}^\star} \mathrm{cost}(\mathbf{w}_P^*)\\ =& O(\sqrt{\lambda}) \cdot \mathrm{cost}(\mathcal{P}^*).\end{aligned}$$

In particular, the two *minimum spanning trees* that obtains $\mathrm{mst}_{s,t}(P)$ gives a feasible walk for serving requests in P as follows: Let $(s,t) = \arg\min_{(s_i,t_i)\in P} d(s_i,t_i)$. Pick any $s' \in \{s_r : r \in P\}, s' \ne s$, and by the classical Christofide's algorithm we easily find a s'-s TSP path on $\{s_r : r \in P\}$ with cost at most $O(\mathrm{mst}_s(P))$. Similarly, we can find a t-t' TSP path on $\{t_r : r \in P\}$ with cost at most $O(\mathrm{mst}_t(P))$. By gluing the two TSP path together through (s,t) we get a feasible walk, $\mathbf{w}_P$, on P with cost $O(\mathrm{mst}_{s,t}(P) + \min_{r_i \in P} d(s_i,t_i))$. This is the $\mathbf{w}_P$ used in the theorem statement. Since $\mathrm{cost}(\mathcal{P}) \le \sum_{P\in\mathcal{P}} \mathrm{cost}(\mathbf{w}_P)$, the existence of $\mathbf{w}_P$ also implies $\mathrm{cost}(\mathcal{P}) \le O(\sqrt{\lambda}) \cdot \mathrm{cost}(\mathcal{P}^*)$. □

6 PART (II): ROUTING

After invoking Algorithm 2 to get the partition $\mathcal{P}$ of requests, we now describe how to find actual routes for the vehicles — the *assignment* $\mathcal{A}$. The requests will be served group-by-group, meaning that each group is served *exclusively* and *non-preemptively*[4] by some vehicle. Such route is of course unlikely to be optimal, nevertheless, the previous hierarchical grouping phase provides a nice structure, which allows us to prove a good approximation ratio.

Our routing phase consists of two steps: First we will generate a set of graphs, specifically we call a *rooted spanning forest* (defined below) $\mathcal{F}$ that connects each group to exactly one vehicle in V_K; Then we design a routing plan that schedules the vehicles to serve its connected groups along the edges of $\mathcal{F}$. We now describe the algorithm formally.

(I). Finding the rooted spanning forest $\mathcal{F}$. First let us formally define the rooted spanning forest.

Definition 5 (Rooted Spanning Forest (RSF)). *Given a weighted graph $G = (V,E)$ with edge cost $c : E \mapsto \mathbb{R}_{\ge 0}$ and a root set $U \subseteq V$, we say a set $\mathcal{F} = \{T_i\}_i$ of (disjoint) trees is a* rooted spanning forest (RSF)*, if*

(1) Each $T_i \in \mathcal{F}$ is a tree rooted at some vertex of U;
(2) $T_i \cap T_j = \emptyset$ for any $i \ne j$;
(3) For any non-root $v \in V \setminus U$, there is some $T_i \in \mathcal{F}$ contains v.

Lastly, define the cost of $\mathcal{F}$ as $c(\mathcal{F}) = \sum_{T\in\mathcal{F}} c(T) = \sum_{T\in\mathcal{F}} \sum_{e\in T} c(e)$. We say $\mathcal{F}$ is a Minimum Rooted Spanning Forest (MRSF) *if $\mathcal{F}$ achieves minimum cost among all rooted spanning forests.*

Claim 1. *Given G, c, U as above in Definition 5, we can find a minimum rooted spanning forest (MRSF) $\mathcal{F}$ in polynomial time.*

Proof. We first contract U to a single vertex u_0. Then for any non-root vertex v, we merge all parallel edges between v and u_0, and re-define the cost $c(u_0,v) := \min_{x\in U} c(x,v)$. Then we find a minimum spanning tree T in this contracted graph (using, e.g., Prim's algorithm). Now we un-contract u_0 back to U: if a non-root vertex v was connected to u_0 by T, then after un-contraction it is connected to its closest neighbor in U.

It is easy to see that after un-contraction T becomes a rooted spanning forest (with the same cost) in the original graph G, which is our desired solution $\mathcal{F}$. To see that $\mathcal{F}$ is minimum: suppose for contradiction there is another rooted spanning forest $\mathcal{F}'$ with smaller cost, then if we contract U, $\mathcal{F}'$ gives a spanning tree cheaper than T, contradicting with T being minimum. □

We will build a minimum rooted spanning forest for the groups $\mathcal{P}$ with root set V_K (the vehicles). Formally, consider the complete graph over vertices $V_K \cup \mathcal{P}$, with V_K being the root set. We define the edge cost c on this graph as follows: recall d is the underlying metric, and let c be

$$\left.\begin{aligned} c(P,P') &:= \min_{r_i\in P, r_j\in P'} d(s_i,s_j), && P,P' \in \mathcal{P} \\ c(u,P) &:= \min_{r_i\in P} d(u,s_i), && u \in V_K, P \in \mathcal{P} \\ c(u,v) &:= d(u,v), && u,v \in V_K \end{aligned}\right\} \tag{1}$$

Now, using Claim 1 we find a minimum rooted spanning forest $\mathcal{F}$ w.r.t. the cost c above. Note that each tree $\mathcal{T} \in \mathcal{F}$ contains *exactly* one vertex from V_K, which we will designate as the *root* of $\mathcal{T}$.

The above process is summarized in Algorithm 3.

Algorithm 3 MRSF($V_K, \mathcal{P}$)

```
Input: Vehicle locations $V_K$ and partition $\mathcal{P}$
Output: A rooted spanning forest $\mathcal{F}$ on $V_K \cup \mathcal{P}$.
1: Let $c$ be given as in Eq (1)
2: Find a minimum rooted spanning forest $\mathcal{F}$ on $V_K \cup S$ with cost function
   $d$, based on Claim 1.
3: return $\mathcal{F}$
```

(II). DFS on $\mathcal{F}$ to serve all requests. For each tree $\mathcal{T} \in \mathcal{F}$, all of its request will be served using only the unique vehicle that is located at the root of $\mathcal{T}$. So, now we can focus on serving a single tree $\mathcal{T}$. Roughly speaking, the vehicle leaves the root of $\mathcal{T}$ and serves each group in a *depth-first* manner along the edges of $\mathcal{T}$. But since each group contains multiple locations, the apparent question is, how exactly does a vehicle move?

Let $S(P)$ denote all the pickup locations from group P. Recall that, by construction each edge (P,P') (or $(p_k,P), p_k \in V_K$) in $\mathcal{T}$ uniquely corresponds to an edge (s,s') (resp. (p_k,s)) for some $s \in S(P)$ and $s' \in S(P')$, so we can think of P, P' are connected via the "portals" s, s'. We also denote the portal via which a group P connects to its parent as $s_0(P)$. For ease of presentation, we also define $s_0(p_k) := p_k$ for all $p_k \in V_K$. The vehicle will always enter P at $s_0(P)$ from its parent.

Then, let $\mathbf{w}_P$ be the walk serving P that is guaranteed by Theorem 2, and let s_1 be the starting point of $\mathbf{w}_P$. The vehicle will first move to s_1 from $s_0(P)$ and serve all requests of P by following $\mathbf{w}_P$, then traverse $S(P)$ again in the order determined by $\mathbf{w}_P$, serving its children groups recursively. Finally, the vehicle move back to $s_0(P)$ and return to P's parent. The process is summarized in Algorithm 4. We also provide an example in Figure 3.

With Algorithm 3 (Mrsf) and 4 (Dfs) at hand, the final Routing algorithm is quite straightforward: just apply Dfs to each tree of the rooted spanning forest returned by Mrsf, as shown in Algorithm 5.

7 PROOF OF THEOREM 1

Now we can prove our main theorem, restated below.

Theorem 1. *Given a Multi-Vehicle DaRP with set of requests R and set of vehicles K, each with a capacity λ, the HGR algorithm runs in time $O(|R|^3 \log\lambda + |R|^2\lambda^2 \log\lambda)$ and returns a set of $|K|$ feasible walks serving all requests in R such that the total travel distance is at most $O(\sqrt{\lambda} \cdot \log|R|)$ times that of an optimal solution to the Muil-Vehicle DaRP.*

[4]By *non-preemptive*, we mean a vehicle must finish serving all requests of a group before it can start serving other groups.

Algorithm 4 DFS(P, $\mathcal{T}$)

```
Input: $P \in V_K \cup \mathcal{P}$, and $\mathcal{T}$ is a tree on $V_K \cup \mathcal{P}$ containing $P$
Output: A feasible walk $\mathbf{w}$ covering the subtree of $\mathcal{T}$ rooted at $P$
1:  $\mathbf{w} \leftarrow \langle s_0(P) \rangle$
2:  if $P \in \mathcal{P}$ then
3:      $\mathbf{w}_P \leftarrow$the walk guaranteed by Theorem 2
4:  else
5:      $\mathbf{w}_P \leftarrow \langle P \rangle$ // If $P \in V_K$ is a vehicle location
6:  $s \leftarrow$starting point of $\mathbf{w}_P$
7:  Append $\mathbf{w}_P$ to $\mathbf{w}$ // Serve $P$
8:  for each $s \in S(P)$ in the order of $\mathbf{w}_P$ do
9:      for each child group $P'$ of $P$ connected via $s$ do
10:         Append $s$ to $\mathbf{w}$
11:         $\mathbf{w}' \leftarrow$DFS($P'$, $\mathcal{T}$)
12:         Append $\mathbf{w}'$ to $\mathbf{w}$
13:     Append $s$ to $\mathbf{w}$
14: Append $s_0(P)$ to $\mathbf{w}$
15: return $\mathbf{w}$
```

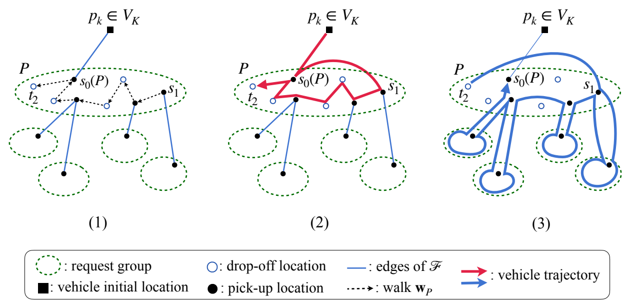


Figure 3: Example of serving the groups using DFS. Figure (1) shows the constructed spanning forest $\mathcal{F}$, and $\mathbf{w}_P$ that starts at some s_1 and ends at t_2. Figure (2) shows how the vehicle serves P: it enters P at $s_0(P)$, then move to s_1 and serves P by following $\mathbf{w}_P$. Figure (3) shows how the vehicle recursively serve P's children after serving P: it first moves back to s_1, and visit children of P in the order of $\mathbf{w}_P$ (breaking ties arbitrarily).

Algorithm 5 ROUTING(R, $\mathcal{P}$, V_K)

```
Input: $R$, $\mathcal{P}$ and $V_K$
Output: An assignment $\mathcal{A}$ that serves $R$
1: $S \leftarrow$ set of all pick-up locations of $R$.
2: $\mathcal{F} \leftarrow$ MRSF($V_K$, $\mathcal{P}$) // Proc. 3
3: $\mathcal{A} \leftarrow \emptyset$
4: for each tree $\mathcal{T} \in \mathcal{F}$ do
5:     $p_k \leftarrow$root of $\mathcal{T}$
6:     $\mathbf{w}_\mathcal{T} \leftarrow$ DFS($p_k$, $\mathcal{T}$) // Proc. 4
7: return $\mathcal{A} = \{\mathbf{w}_\mathcal{T} : \mathcal{T} \in \mathcal{F}\}$
```

Let $\mathcal{A}$ be the solution returned by HGR (Algorithm 1), $\mathcal{P}$ be the partition output by HIERARCHICAL GROUPING (Algorithm 2), and $\mathcal{F}$ be the rooted spanning forest obtained by MRSF(V_K, $\mathcal{P}$) (Procedure 3).

First, we have the following simple claim on the cost of $\mathcal{F}$.

Claim 2. *Let S denote the set of all pick-up locations in R. For any request partition $\mathcal{P}$, let $\mathcal{F}$ = MRSF(V_K, $\mathcal{P}$) (Procedure 3) and* $\mathrm{cost}(\mathcal{F}) := \sum_{e\in\mathcal{F}} d(e)$*, then we have:*

$$\mathrm{cost}(\mathcal{F}) \le \mathrm{cost}(\mathcal{A}^*),$$

where $\mathcal{A}^*$ is any feasible solution to the original Dial-a-Ride problem.

PROOF. The solution $\mathcal{A}^*$, being a collection of walks, can also be thought as a rooted spanning forest over all the pick-up (as well as the drop-off) locations: View each walk as a sequence of weighted edges (with weight given by metric d), then every $s \in S$ is connected to some vehicle from V_K as a root. Now we modify $\mathcal{A}^*$ in three steps to make it also a RSF on $V_K \cup \mathcal{P}$ using the same cost function c (Eq (1)):

(1) Shortcut every drop-off locations to make $\mathcal{A}^*$ a rooted spanning forest on S only. Since the distance d is a metric, this only reduces $\mathcal{A}^*$'s cost;
(2) Then we contract each group of $\mathcal{P}$ to a single vertex, which preserves only between-group edges of $\mathcal{A}^*$. This again only reduces its cost;
(3) Finally, since each remaining edge is either between two groups or between a group and a vertex in V_K, we can reassign its cost using c. By definition of c, this only reduces the total cost.

Denote the resulting graph as $\mathcal{A}'$. By construction, it is a rooted spanning subgraph over $V_K \cup \mathcal{P}$ with at most the same cost of $\mathcal{A}^*$. Then by definition we have $\mathrm{cost}(\mathcal{F}) \le \mathrm{cost}(\mathcal{A}') \le \mathrm{cost}(\mathcal{A}^*)$. □

Now we give the proof for the main theorem.

PROOF OF THEOREM 1. Firstly, any optimal solution $\mathcal{A}'$ is also a collection of walks starting from locations in V_K. Applying Fact 1 to each walk of $\mathcal{A}'$, we get a new solution $\mathcal{A}^*$ such that (1) every walk of $\mathcal{A}^*$ serves requests in a group-by-group manner, where each group is of size at most λ; and (2) $\mathrm{cost}(\mathcal{A}^*) \le \mathrm{cost}(\mathcal{A}') \cdot O(\log n) = O(\log n)\mathrm{OPT}$.

Then we show $\mathrm{cost}(\mathcal{A})$ can be bounded by $O(\sqrt{\lambda}) \cdot \mathrm{cost}(\mathcal{A}^*)$, which will give $\mathrm{cost}(\mathcal{A}) \le O(\sqrt{\lambda}\log n) \cdot \mathrm{OPT}$. $\mathrm{cost}(\mathcal{A})$ can be decomposed into two parts: the cost of traveling along edges in $\mathcal{F}$, and the cost of moving within each group. By the nature of DFS, each edge of $\mathcal{F}$ is traversed exactly twice, therefore the first part of the cost is at most $2\mathrm{cost}(\mathcal{F}) \le 2\mathrm{cost}(\mathcal{A}^*)$ by Claim 2.

For the second part of cost, we fix a request group P and consider the total travel distance of the vehicle within P. Recall $s_0(P)$ is the "portal" connecting P with its parent, and $\mathbf{w}_P$ is the walk given by Theorem 2 that serves all requests in P. Let s' be the starting location of $\mathbf{w}_P$. The vehicle first moves from $s_0(P)$ to s', which takes at most $\mathrm{cost}(\mathbf{w}_P)$. It then serves all of P by following $\mathbf{w}_P$, which takes another $\mathrm{cost}(\mathbf{w}_P)$. The vehicle then moves back to s' and traverse $\mathbf{w}_P$ again to recursively serve all children groups of P, and finally moves to $s_0(P)$. This process will cost at most another $3\mathrm{cost}(\mathbf{w}_P)$. So overall the travel distance within P is at most $5\mathrm{cost}(\mathbf{w}_P)$. (Note this is apparently not the most efficient moving strategy, but it suffices to give the desired bound)

To summarize, $\mathrm{cost}(\mathcal{A}) \le 2\mathrm{cost}(\mathcal{A}^*) + 5\sum_{P\in\mathcal{P}} \mathrm{cost}(\mathbf{w}_P) \le O(\sqrt{\lambda}) \cdot \mathrm{cost}(\mathcal{A}^*)$, where the last inequality is by Theorem 2. This concludes our proof. □

8 COMPUTATIONAL EXPERIMENTS

8.1 DaRP Instances and Baselines

Synthetic datasets. We first benchmark HGR on two synthetic datasets, in order to gain insights about the overall solution quality and runtime performance. In the first dataset (SY-U), locations (i.e., request pickups, drop-offs and drivers' initial locations) are randomly generated from a uniform distribution on a $[0, 100]^2$ grid. In the second dataset (SY-G), locations are randomly generated from a Gaussian mixed-model (GMM) with Z clusters. Each cluster corresponds to a bivariate Gaussian distribution whose center is drawn from a uniform distribution on a $[0, 1000]^2$ grid and with covariance matrix given by $\sigma^2\mathbf{I}$. SY-G instances represent a topography with multiple demand clusters as typically found, e.g., in sparse urban areas. For the SY-G distribution, several combinations of parameters Z and σ are tested, as detailed in Table 1. The choices of these synthetic distributions mainly follow the same setting used in previous works [37, 38].

Realworld datasets. We also test the algorithms on two realworld datasets consisting of transportation data from New York City (NYC) and San Francisco (SFO).

- NYC: We use NYC Taxi & Limousine Commission Trip Record Data [30]. In particular, we randomly select 10,000 trip records from May/2016.
- SFO: We use the Cab Spotting Data [34], which records roughly 500 taxis' trace data in a period of 30 days. Again, we randomly select 10,000 trip records from the original dataset.

In these datasets a location is specified by its latitude and longitude coordinates. The distance between two locations is defined to be the graph (shortest-path) distance calculated via the actual road map of the two cities, which we obtain from the OpenStreetMap database [31]. All parameter settings are detailed in Table 1.

Table 1: Parameter settings for the datasets. Bold values indicate fixed parameter values for the sensitivity analyses.

	SY-U, NYC and SFO	SY-G
n	2, 4, 6, **8**, 10 ($\times 10^3$)	2, 4, **6** ($\times 10^3$)
m	30, 60, **90**, 120, 150	30, 60, **90**
λ	2, 4, 8, 16, **32**, 64	4, 8, **16**
Z		5, **10**, 20, 50, 100
σ		5, 10, 25, **50**, 100, 250

Metrics and Baselines. We measure the performance of the algorithms with respect to two objectives. The first is the *total travel distance*, which is what HGR is set to optimize. The second objective is the *total in-transit latency*: the in-transit latency for a rider is the time length he/she is on board, i.e., the time between being picked-up and finally dropped-off at his/her destination. Total in-transit latency corresponds to the sum of in-transit latency over all rides. This objective is crucial for customer experience as most riders prefer to reach their destination as quickly as possible after being picked-up.

We adopt two recent DaRP algorithms as the main baselines in our experiments:

- pruneGDP [37]: This is a heuristic algorithm that builds routes incrementally by greedily inserting new requests. It is designed to optimize the total travel distance and can be implemented very efficiently, though no approximation guarantee is known.
- FESI [38]: This algorithm optimizes the *makespan* of vehicles, i.e., the maximum distance traveled by the vehicles. However, as claimed in [38], FESI also obtains small total travel distance on various datasets, often comparable with pruneGDP. We remark that FESI has a $O(\sqrt{\lambda}\log n)$ approximation guarantee in terms of the makespan objective, but no guarantee for total travel distance is known.

There are many classical (meta-)heuristic and exact algorithms for DaRP, including Branch-and-Bound [6, 16, 35], Tabu Search [7, 10], Neighborhood Search [15, 32], and Genetic Algorithms [8, 22, 29]. Compared with these algorithms, the two baselines we choose stand out particularly with their ability to handle large instances: both of them can deal with tens of thousands of requests very efficiently, while most former algorithms can only handle a few thousand requests at most. Besides, FESI and pruneGDP achieved state-of-the-art performance in previous experiments [37, 38]. We also refer to a recent and comprehensive benchmark of heuristic algorithms [18].[5]

Implementation. We use the publicly-available code provided by [38] for FESI. Other algorithms (including ours) are implemented in C++. All experiments are conducted on a single core of an Intel® Xeon® Gold 6130 (2.1GHz) processor with 32GB of available RAM. As FESI is a randomized algorithm, we run FESI 10 times on each instance and report the average results. Our implementations, with which all results can be reproduced, is publicly available at `https://github.com/amflorio/hga-dial-a-ride`.

8.2 Computational Results

Figures 4 — 6 depict the results on synthetic datasets. Results on realworld datasets are shown in Figure 7 and 8. Overall, our algorithm exhibits clear superiority on both objectives in almost all parameter regimes and datasets. Now, we discuss the effect of each parameter in more details.

Synthetic datasets. Figures 4 and 5 illustrate the effect of n, m and λ on the algorithms' performance, for data generated from uniform and GMM distributions, respectively. We remark that the trend on the synthetic datasets are much similar to that on the realworld datasets (Figures 7 and 8), therefore we postpone a detailed discussion on the effect of n, m and λ to the later part when reporting results on realworld datasets. Generally speaking, for the total travel distance objective, our algorithm HGR consistently performs the best. When it comes to in-transit latency, our algorithm still achieves the best objective in most parameter regimes, but FESI is able to exploit more or larger vehicles, and is likely to provide better latency.

[5]The benchmark results can also be accessed at `https://sites.google.com/site/darpsurvey/comparison`.

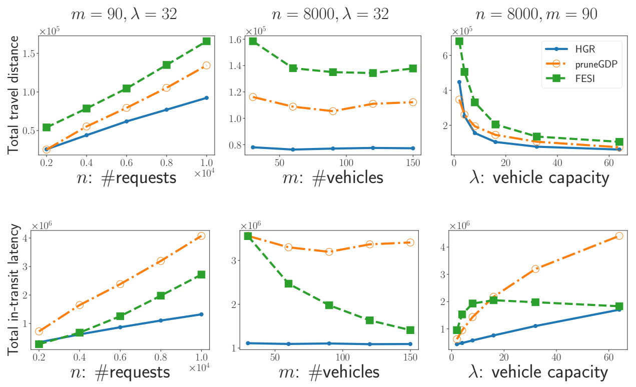


Figure 4: Results on the Uniform synthetic dataset (SY-U) with varying n, m, λ.

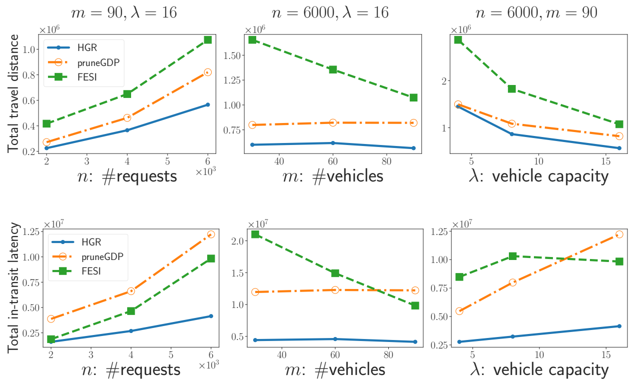


Figure 5: Results on the GMM synthetic dataset (I): varying n, m, λ.

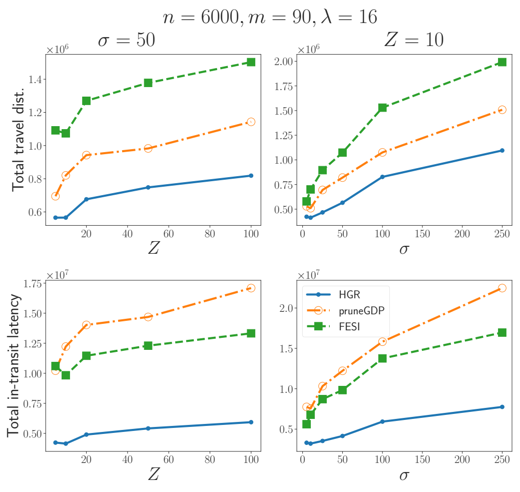


Figure 6: Results on the GMM synthetic dataset (II): varying Z and σ.

Figure 6 shows how the distribution parameters (specifically, the number of clusters and covariance of the GMM) affects algorithm performance. The first two columns of Figure 6 show the effect of varying the number of clusters (Z) of the GMM, and the remaining two show the effect of varying σ. Generally speaking, the more spread-out the data are, the larger advantage our algorithm has: when σ is very small, the performance of all three algorithms are very close to each other on both objectives. This is expected as all the requests are highly concentrated around only $Z = 10$ centers, which makes good choices of routes very limited. On the other hand, when σ is larger (which has the similar effect as larger Z with fixed σ), our algorithm exhibits clear superiority, with over 50% less total in-transit latency than FESI or pruneGDP, and 30% less total travel distance. HGR is also much less sensitive to the change of Z or σ compared with the two baselines.

Realworld data. Concerning the effect of n, m and λ, results on the two realworld datasets as well as the synthetic datasets are quite similar, so we will take one of them as example. Figure 7 shows the result on the NYC dataset. Both the total travel distance and in-transit latency grow with n, as expected, and the gap between HGR and the baselines also grows with n (the first column of Figure 7). Notice that FESI performs better than pruneGDP in terms of in-transit latency, because it explicitly optimizes makespan and results in shorter per-vehicle trips.

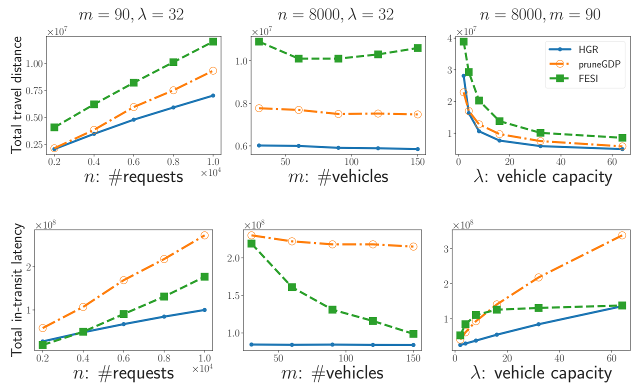


Figure 7: Results on the NYC dataset.

The more interesting part is when we fix n and λ, and vary the number of vehicles m (the second column of Figure 7). Notice that the latency of our algorithm (HGR) is almost unaffected by m. This occurs because the first phase (i.e., Algorithm 2) of HGR is independent of vehicle locations. After the requests have been partitioned into groups, the in-transit latency is essentially determined no matter how we assign vehicles to these groups. A surprising fact is that the total travel distance of HGR is also little affected by m. After inspecting the actual routes generated by HGR, we find that although there are many vehicles available, the minimum spanning forest $\mathcal{F}$ found in Algorithm 5 uses only a few vehicles.

The third column of Figure 7 shows the result where we fix n and m and vary λ. Larger λ generally leads to less travel distance, since larger capacity allows more flexible choices of routes. Our

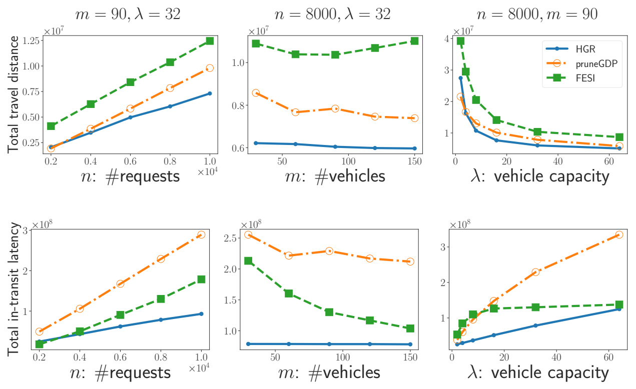


Figure 8: Results on the SFO dataset.

algorithm still outperforms the two baselines in both objectives, but from the last plot one can expect FESI to have better latency when λ is larger. This is, again, because FESI aims to optimize makespan, thus larger λ does not necessarily lead to longer per-vehicle routes, while it is the opposite for HGR and pruneGDP.

We remark that, for both the synthetic datasets and the realworld datasets, the effect of varying n, m and λ are much the same.

Running Time Analysis. In the Grouping phase, we construct at most n^2 minimum spanning trees in each iteration $\ell < \log \lambda$, each of them can be constructed in time $O(2^{2\ell} \log(2^\ell))$ [33]. The most time-consuming part is step 8 where we compute a minimum-cost perfect matching in each iteration, and the currently best-known algorithm takes time $O(n^3)$ [13] in a complete graph. Since there are $\log \lambda$ iterations, the running time of Grouping phase is $O(n^3 \log \lambda + n^2\lambda^2 \log \lambda)$. In the Routing phase, we find minimum spanning forest in time $O(n^2 + m)$ and route the walks in time $O(mn)$. Obviously, $m \le n$. In total, the running time is $O(n^3 \log \lambda + n^2\lambda^2 \log \lambda)$.

In our actual implementation of HGR, we use the Blossom V [23] matching algorithm due to its widespread use in practice, in spite of having a slightly worse theoretical guarantee. In the experiments, for inputs consisting of 10,000 requests and 150 vehicles with capacity 64, our algorithm takes less than 800 seconds to finish. In a more practical instance with 4,000 requests and 90 vehicles with capacity 8, HGR takes about 80 seconds.

Although reasonably fast, our basic implementation of HGR turns out to be significantly slower than FESI and pruneGDP, both of which have only an $O(n^2)$ dependence on the number of requests n. For example, on the largest input mentioned above (n=10,000, m=150, λ=64), our algorithm (HGR) is 20 times slower than FESI and pruneGDP. In Section 9, we show how to implement a much more scalable version of HGR by replacing several components of the vanilla algorithm with their *approximate versions*, while sacrificing slightly the solution quality.

9 SCALABILITY WITH LARGE INSTANCES

9.1 The HGR-approx Algorithm

As discussed at the end of Section 8, the dominating factor of the running time comes from the matching step (Line 8 of Algorithm 2). There exists near linear-time (i.e., $O(n^{2+\epsilon})$, because the input here can have $O(n^2)$ edges) algorithms that output a near-optimal perfect matching [11], which in principle can reduce our algorithm's running time to $O(n^2)$ (assuming λ being a relatively small constant) with some negligible loss in approximation ratio. However, such matching algorithms are sophisticated and not easy to implement in practice.

We therefore resort to simpler *approximations*: instead of finding the min-cost perfect matching, we find a non-optimal matching using a "bucketing" method (details see below). Besides, the "edge cost" w (Definition 4) used in our graph requires computing multiple MST over the two clusters, which is quite costly. Specifically, those MST computations come from evaluating $w_1(\cdot,\cdot)$ between groups. We therefore replace w_1 with a simpler cost function that approximates it. Specifically, we implement the two following approximate versions of HGR:

- HGR-w_1: We define a new cost w'_1 between any two groups X and X' as

$$w'_1(X, X') := \min_{r_i \in X, r_j \in X'} d(s_i, s_j) + \min_{r_i \in X, r_j \in X'} d(t_i, t_j).$$

  The new algorithm HGR-w_1 still finds the minimum-cost *perfect* matching (like HGR), but uses w'_1 in place of w_1.
- HGR-approx: This algorithm builds upon HGR-w_1. In addition to using w'_1 in place of w_1, HGR-approx finds a perfect matching using a "bucketing" heuristic. Suppose the largest edge cost is Δ. We first divide all edges into $O(\log \Delta)$ buckets, where the i-th bucket contains all edges with cost $[(1+\delta)^{i-1}, (1+\delta)^i)$, where δ is a small constant. Then, for each bucket we compute a *maximal* matching using only the edges of the bucket.
  The maximal matching is computed using a simple greedy method. Starting with an empty matching, the algorithm greedily chooses the lowest-cost edge that is disjoint with the current matching and includes it into the solution. It is straightforward to implement this heuristic in $O(n^2 \log \Delta)$ time.

The HGR-w_1 algorithm still has an $O(n^3)$ dependence with n since it needs to find a perfect matching, though it avoids the $O(n^2\lambda^2 \log \lambda)$ additive factor. The HGR-approx algorithm instead runs in $O(n^2 \log \Delta)$ time.

9.2 Experimental Evaluation

We inherit most of the experiment settings from Section 8, and test HGR-approx on much larger instances. The results are quite similar, so we only plot some representative results on the NYC dataset. We are still comparing HGR-approx with pruneGDP and FESI. To examine how the approximation affects solution quality, we also include comparison with the original HGR in some smaller instances as in Section 8. (The original HGR algorithm is too slow for instance as large as $n = 10^5$ and does not terminate in reasonable time.)

Figure 9 shows the results on large instances with up to 100k requests and 10k drivers. One can see that HGR-approx still achieves

Table 2: Parameter setting for the larger datasets.

n (#requests)	2.5, 5, **10** ($\times 10^4$)
m (#vehicles)	1, 2.5, 5, **10**, ($\times 10^3$)
λ (vehicle capacity)	2, 4, 8, 16, **32**, 64

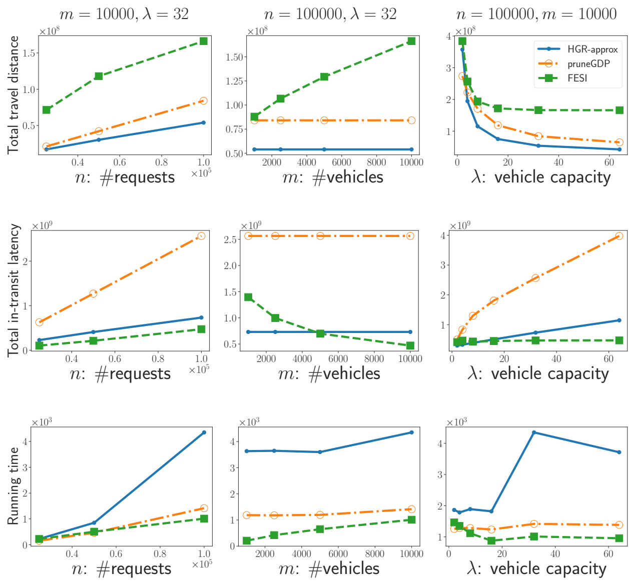


Figure 9: Results on large-scale inputs from the NYC dataset.

best total travel distance (the first row), while having comparable in-transit latency (the second row) with the state-of-the-art benchmark (FESI). Note that FESI explicitly optimizes makespan (the largest travel distance of all vehicles), which often also leads to small in-transit distance since the solution is formed by many short trips, but the total distance can be very large. The running time (the third row) of HGR-approx is, however, still larger than the two benchmarks, though the difference ($\sim 4\times$) is much smaller than that of the original HGR, which does not even finish in a reasonable time on such-sized instances. This indicates that our algorithm has the potential to scale to large instances.

Figure 10 illustrates the performance of HGR-approx on smaller instances, with comparison to the original HGR algorithm and HGR-w_1. One can see that the approximations do lead to (slight) performance degradation. In terms of total travel distance (which is the objective our algorithm optimizes), HGR-approx is slightly worse than HGR-w_1, which is slightly worse than HGR. The effect on in-transit latency is even less obvious. The approximation also greatly reduces the running time of HGR, though they are still higher than pruneGDP and FESI.

In summary, we are able to accelerate HGR significantly using some straightforward approximation or heuristics, sacrificing the solution quality only slightly. The main ingredient of the algorithm — hierarchically grouping requests — is quite flexible and provides a good starting point to apply other routing methods.

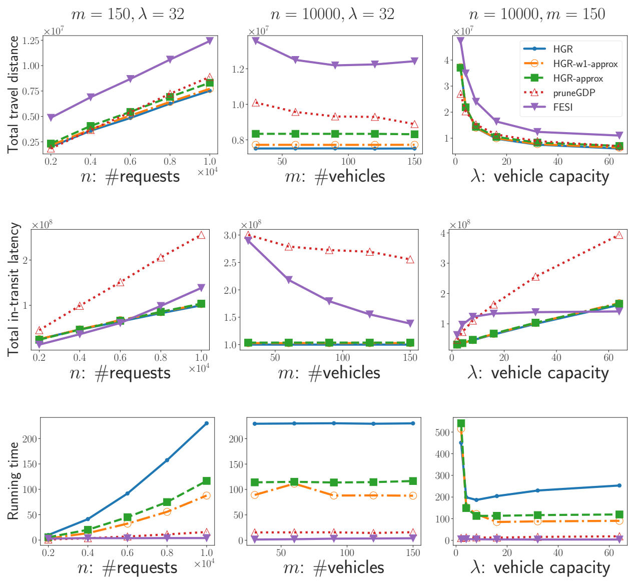


Figure 10: Results on inputs from NYC dataset of the same scale as Section 8.

10 DISCUSSION

In this paper we propose an algorithm for the multi-vehicle Dial-a-Ride problem with the objective to optimize the total travel distance of all vehicles. The $O(\sqrt{\lambda}\log n)$ approximation ratio of our algorithm matches that of the best known algorithm for the single-vehicle case. It is still an open problem whether this ratio can be improved even in the single-vehicle case. We provide three different implementations of the basic algorithm with increasing runtime efficiency. We experimentally demonstrate that all versions of our algorithm outperform two recent state-of-the art heuristics for this problem on both synthetic and real world datasets. Further, we showcase scalability of the most efficient implementation to datasets of size up to 100,000 requests.

Our work is a significant first step towards building theoretically sound algorithms for multi-vehicle Dial-a-Ride that are also practical. There are several intriguing open questions that out work raises. Firstly, our algorithm does not directly handle deadline constraints that are often encountered in practice. Note that we could heuristically incorporate deadlines using the ideas given in [38]: first, solve the problem without deadlines using HGR, and then use the insertion subroutine from [37] as long as no deadline constraint is violated. However, proving similar approximation guarantees as in this work becomes much more challenging in this case. We leave this as a future research direction.

By replacing various components of our algorithm with their approximate versions, we are able to greatly boost its scalability. We believe that the runtime can be further improved by exploiting the inherent parallelism of many of the steps (in particular, the capacitated grouping part) and utilizing algorithms developed in the recently popular Map-Reduce models of computation [24]. We leave this as our second open problem.

REFERENCES

[1] Xiaohui Bei and Shengyu Zhang. 2018. Algorithms for Trip-Vehicle Assignment in Ride-Sharing. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. 3–9.

[2] Hua Cai, Xi Wang, Peter Adriaens, and Ming Xu. 2019. Environmental benefits of taxi ride sharing in Beijing. *Energy* 174 (2019), 503–508.

[3] Moses Charikar and Balaji Raghavachari. 1998. The finite capacity dial-a-ride problem. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS'98), November 8-11, 1998, Palo Alto, California, USA*. 458–467.

[4] Peng Cheng, Hao Xin, and Lei Chen. 2017. Utility-Aware Ridesharing on Road Networks. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. New York, NY, USA, 1197–1210.

[5] Regina R Clewlow and Gouri S Mishra. 2017. Disruptive transportation: The adoption, utilization, and impacts of ride-hailing in the United States. (2017). https://escholarship.org/uc/item/82w2z91j

[6] Jean-François Cordeau. 2006. A Branch-and-Cut Algorithm for the Dial-a-Ride Problem. *Oper. Res.* 54, 3 (2006), 573–586.

[7] Jean-François Cordeau and Gilbert Laporte. 2003. A Tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological* 37 (07 2003), 579–594.

[8] Claudio Cubillos, Enrique Urra, and Nibaldo Rodríguez. 2009. Application of genetic algorithms for the DARPTW problem. *International Journal of Computers Communications & Control* 4, 2 (2009), 127–136.

[9] Willem E. de Paepe, Jan Karel Lenstra, Jiri Sgall, René A. Sitters, and Leen Stougie. 2004. Computer-Aided Complexity Classification of Dial-a-Ride Problems. *INFORMS Journal on Computing* 16, 2 (2004), 120–132.

[10] Paolo Detti, Francesco Papalini, and Garazi Zabalo Manrique de Lara. 2017. A multi-depot dial-a-ride problem with heterogeneous vehicles and compatibility constraints in healthcare. *Omega* 70 (2017), 1–14.

[11] Ran Duan and Seth Pettie. 2010. Approximating maximum weight matching in near-linear time. In *Proceedings of the 51th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2010), October 23-26, 2010, Las Vegas, Nevada, USA*. 673–682.

[12] Jack Edmonds. 1965. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of research of the National Bureau of Standards B* 69, 125-130 (1965), 55–56.

[13] Harold N Gabow. 1990. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1990), 22-24 January 1990, San Francisco, California, USA*. 434–443.

[14] Thierry Garaix, Christian Artigues, Dominique Feillet, and Didier Josselin. 2010. Vehicle routing problems with alternative paths: An application to on-demand transportation. *European Journal of Operational Research* 204, 1 (2010), 62–75.

[15] Timo Gschwind and Michael Drexl. 2019. Adaptive large neighborhood search with a constant-time feasibility test for the dial-a-ride problem. *Transportation Science* 53, 2 (2019), 480–491.

[16] Timo Gschwind and Stefan Irnich. 2015. Effective handling of dynamic time windows and its application to solving the dial-a-ride problem. *Transportation Science* 49, 2 (2015), 335–354.

[17] Anupam Gupta, MohammadTaghi Hajiaghayi, Viswanath Nagarajan, and Ramamoorthi Ravi. 2010. Dial a ride from k-forest. *ACM Transactions on Algorithms (TALG)* 6, 2 (2010), 1–21.

[18] Sin C Ho, Wai Yuen Szeto, Yong-Hong Kuo, Janny MY Leung, Matthew Petering, and Terence WH Tou. 2018. A survey of dial-a-ride problems: Literature review and recent developments. *Transportation Research Part B: Methodological* 111 (2018), 395–421.

[19] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. 2014. Large scale real-time ridesharing with service guarantee on road networks. *Proceedings of the VLDB Endowment* 7, 14 (2014), 2017–2028.

[20] Jang-Jei Jaw. 1984. Solving large-scale dial-a-ride vehicle routing and scheduling problems. *FTL report (Massachusetts Institute of Technology. Flight Transportation Laboratory)* (1984). https://hdl.handle.net/1721.1/68045

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

[22] Rene Munch Jorgensen, Jesper Larsen, and Kristin Berg Bergvinsdottir. 2007. Solving the dial-a-ride problem using genetic algorithms. *Journal of the operational research society* 58, 10 (2007), 1321–1331.

[23] Vladimir Kolmogorov. 2009. Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation* 1, 1 (2009), 43–67.

[24] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. 2011. Filtering: A Method for Solving Graph Problems in MapReduce. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11), San Jose, CA, USA, June 4-6, 2011* (San Jose, California, USA). New York, NY, USA, 85–94.

[25] Kelin Luo, Chaitanya Agarwal, Syamantak Das, and Xiangyu Guo. 2022. The Multi-vehicle Ride-Sharing Problem. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining (WSDM'22), Virtual Event / Tempe, AZ, USA, February 21 - 25, 2022*. 628–637.

[26] Kelin Luo, Alexandre M Florio, Syamantak Das, and Xiangyu Guo. 2022. A Hierarchical Grouping Algorithm for the Multi-Vehicle Dial-a-Ride Problem. *arXiv preprint arXiv:2210.05000* (2022). https://arxiv.org/abs/2210.05000

[27] Kelin Luo and Frits CR Spieksma. 2020. Approximation algorithms for car-sharing problems. In *Proceedings of the 26th International Conference on Computing and Combinatorics (COCOON 2020), Atlanta, GA, USA, August 29-31, 2020 (Lecture Notes in Computer Science)*, Vol. 12273. Springer, 262–273.

[28] Shuo Ma, Yu Zheng, and Ouri Wolfson. 2013. T-share: A large-scale dynamic taxi ridesharing service. In *2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*. 410–421.

[29] Mohamed Masmoudi, Kris Braekers, Malek Masmoudi, and Abdelaziz Dammak. 2016. A Hybrid Genetic Algorithm for the Heterogeneous Dial-A-Ride Problem. *Computers and Operations Research* 81 (12 2016).

[30] NYCTLC. 2020. New York City Taxi & Limousine Commission trip data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[31] OpenStreetMap. 2021. San Francisco and New York City street map data. https://www.openstreetmap.org.

[32] Sophie N Parragh, Karl F Doerner, and Richard F Hartl. 2010. Variable neighborhood search for the dial-a-ride problem. *Computers & Operations Research* 37, 6 (2010), 1129–1138.

[33] Seth Pettie and Vijaya Ramachandran. 2000. An optimal minimum spanning tree algorithm. In *International Colloquium on Automata, Languages, and Programming (ICALP'2000), Geneva, Switzerland, July 9-15, 2000*. 49–60.

[34] Michal Piorkowski, Natasa Sarafijanovic-Djukic, and Matthias Grossglauser. 2009. CRAWDAD dataset EPFL/mobility. https://crawdad.org/epfl/mobility/20090224/cab.

[35] Stefan Ropke, Jean-François Cordeau, and Gilbert Laporte. 2007. Models and branch-and-cut algorithms for pickup and delivery problems with time windows. *Networks: An International Journal* 49, 4 (2007), 258–272.

[36] Alexander Schrijver et al. 2003. *Combinatorial optimization: polyhedra and efficiency*. Vol. A. Springer. 453–458 pages.

[37] Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, Jieping Ye, and Ke Xu. 2018. A Unified Approach to Route Planning for Shared Mobility. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1633–1646.

[38] Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2019. Last-Mile Delivery Made Practical: An Efficient Route Planning Framework with Theoretical Guarantees. *Proceedings of the VLDB Endowment* 13, 3 (2019), 320–333.

[39] Yuxiang Zeng, Yongxin Tong, Yuguang Song, and Lei Chen. 2020. The Simpler the Better: An Indexing Approach for Shared-Route Planning Queries. *Proceedings of the VLDB Endowment* 13, 13 (Sept. 2020), 3517–3530.

[40] Libin Zheng, Lei Chen, and Jieping Ye. 2018. Order Dispatch in Price-Aware Ridesharing. *Proceedings of the VLDB Endowment* 11, 8 (April 2018), 853–865.

[41] Libin Zheng, Peng Cheng, and Lei Chen. 2019. Auction-Based Order Dispatch and Pricing in Ridesharing. In *35th IEEE International Conference on Data Engineering (ICDE 2019), Macao, China, April 8-11, 2019*. 1034–1045.