# The Multi-vehicle Ride-Sharing Problem

Kelin Luo
k.luo@tue.nl
Eindhoven University of Technology & University of Bonn
Eindhoven, Netherlands & Bonn, Germany

Chaitanya Agarwal
chaitanyaagarwal291@gmail.com
New York University
New York, NY, USA

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

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

## ABSTRACT

Ride-sharing is one of the most popular models of economical and eco-friendly transportation in modern smart cities, especially when riding hybrid and electric vehicles. Usually multiple passengers with similar itineraries are grouped together, which significantly reduces travel cost (or time), road congestion, and traffic emissions.

In this paper, we study the ride-sharing problem where each vehicle is shared by exactly $\lambda$ riders for any fixed $\lambda > 0$, and the goal is to minimize the total travel distance. The min-cost ride-sharing problem is intractable even in the case of exactly two riders sharing a vehicle [2], and hence we can only hope for an approximate solution.

We propose a novel two-phase algorithm: a hierarchical grouping phase that partitions requests into disjoint groups of fixed size, followed by an assignment of request groups to individual vehicles and planning a feasible route for each vehicle. This is the first non-trivial approximation algorithm for the ride-sharing problem with vehicle capacity larger than two.

We verify the efficacy of our algorithm on both synthetic and realworld datasets. Our experiments show that, the ride-sharing scheme produced by our algorithm not only has small total travel distance compared to state-of-the-art baselines, but also enjoys a small makespan and total latency, which crucially relate to each single rider's traveling time. This suggests that our algorithm also enhances rider experience while being energy-efficient.

## CCS CONCEPTS

• **Applied computing** → **Transportation**; • **Theory of computation** → **Routing and network design problems**.

## KEYWORDS

ride-sharing, approximation algorithm

**ACM Reference Format:**
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), February 21–25, 2022, Tempe, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3488560.3498449


*WSDM '22, February 21–25, 2022, Tempe, AZ, USA*

ACM ISBN 978-1-4503-9132-0/22/02.
https://doi.org/10.1145/3488560.3498449

## 1 INTRODUCTION

Ride-sharing has grown to be one of the most important aspects of shared-economy [6] in recent years. All major urban taxi-providers like Uber, Lyft or Didi Chuxing have introduced the option to car-pool. Further, carpooling is becoming increasingly popular among cross-country travellers, boosted by platforms like Bla Bla Car [3] and Transvision [19]. On such platforms, travellers sharing similar routes can rent a car and share their long distance commutes. Corporations with offices spread out across a city often provide cab services to their employees and carpooling is an essential aspect of such a service.

Ride-sharing offers economic advantages from the perspective of both the riders and the service providers. Riders can travel at a cheaper cost due to sharing. The transportation provider can better utilize their limited resources and improve their profit margin by attracting more riders. Perhaps more importantly, ride-sharing has a massive positive impact on the environment. It can potentially reduce air pollution by reducing the number of vehicles on road [7] and optimize cumulative fuel consumption leading to reduced carbon footprint [4, 21].

We consider the following scenario. There is a set of riders who have specified their origins and destinations to the vehicle provider platform and the provider has a set of cars. Depending on the context, the vehicles maybe driven by a chauffeur or driven by one of the riders (for instance, carpooling through Bla Bla Car works under this model). The task is to suitably assign riders to vehicles without exceeding the vehicle's capacity and plan a route for each of the vehicles based on a suitable optimization criteria.

Majority of related literature in this area have considered the objectives of maximizing the revenue [11, 18, 23–25] or minimizing the travel distance/time of the vehicles [2, 13, 18, 22]. In this work, we consider the classical objective of minimizing the *total travel distance of all cars*. Our choice of objective is primarily guided by a motivation to provide energy-efficient solutions to the ride-sharing problem. In spite of a rich literature on shared mobility, there is a surprising dearth of principled techniques to tackle the particular problem described above. We point out two major limitations of existing techniques as follows.

**Limitation 1.** The only algorithms with theoretical guarantees known for the ride-sharing problem stated above work for the rather restrictive case where the car capacity is 2 [2, 13]. Very recently, the authors of [23] consider our setting with general capacities, but with a different objective of maximizing the revenue rather than minimizing total travel distance. Hence their algorithms are not applicable for the objective we seek to minimize.

**Limitation 2.** There is some work in the literature which considers a related but subtly different version of the problem [18, 22]. In particular, these papers assume the model where the number of riders assigned to a specific vehicle can exceed the capacity of the vehicle. However, the planned routes ensure that at any point in time, the vehicle carries at most as many riders as its capacity. This is popularly known as the dial-a-ride problem [5, 10]. Recall that our setting requires a more strict allocation where each vehicle is assigned to exactly one group of people and hence the algorithms in the above paper are not directly applicable. Indeed, those algorithms can be modified to solve our problem. However, we are not aware of any theoretical guarantees for such algorithms. Further, our experiments show that such solutions can have 15%-70% higher travel time than our algorithm in practice.

The above discussion makes it clear that the existing solutions for ride-sharing are far from optimal both in theory and practice. In this work, we give the *first approximation algorithm* for the ride-sharing problem minimizing total travel time with vehicle capacity $\lambda \geq 2$. The approximation factor of our algorithm is $O(\sqrt{\lambda})$. In practical scenarios, the value of $\lambda$ is small [1] and hence our algorithm has a small error compared to the optimal solution. At a very high level, our technique follows a *grouping, assigning and route planning* strategy. The first two phases utilize a hierarchical grouping technique in conjunction with minimum cost matching on a suitably defined bipartite graph. A key property of our algorithm is that the 'cost' of the algorithm of these phases can be bounded by a suitably chosen lower bound of the optimal solution. Thanks to the above property, our routing strategy is a fairly straightforward travelling salesman tour which respects the order of pickups and drop-offs. We summarize our **key contributions** as follows.

- We give the first approximation algorithm for the ride-sharing problem minimizing the total travel distance/time that works with any arbitrary vehicle capacities. Our approximation factor is $O(\sqrt{\lambda})$, where $\lambda$ is the capacity of the vehicles.
- We perform extensive experiments on both synthetic and real datasets, where we compare our algorithm with mild adaptations of state-of-the-art algorithms for ride-sharing. Our method outperforms all these algorithms on total travel distance by a significant margin of $40\% - 70\%$ on synthetic and $15\% - 40\%$ on real-world datasets.
- Although our theoretical guarantees are valid only for the objective of minimizing the total travel time, we also perform an empirical evaluation of our method for two other classical objective functions - makespan, that is the maximum time to complete a single trip and total latency, that is sum of service time of each passenger. Surprisingly, we find that, in practice, our algorithm matches or outperforms the state-of-the art algorithms even in terms of the these objectives: 10%-20%-less latency, and 10%-30%-less makespan.

# 2 PROBLEM STATEMENT

## 2.1 Notations and Preliminaries

*Definition 1.* (Distance Metric). Let $V$ be a set of locations. Then the distance metric is defined as the function $d : V \times V \mapsto \mathbb{R}_{\geq 0}$ that satisfies the three properties -

a) (reflexive) $d(v, u) = 0$ if and only if $v = u$
b) (symmetric) $d(v, u) = d(u, v)$
c) (triangle inequality) $d(v, u) \leq d(v, w) + d(w, u)$

Our algorithm works for arbitrary metrics. But in the experiments, we assume the travel distances between two locations to be either the shortest path metric in a suitably defined graph or Euclidean distances.

*Definition 2.* (Request). A request is a tuple $r = \langle s_r, t_r \rangle$ where $s_r \in V, t_r \in V$ are respectively the origin and destination.

*Definition 3.* (Vehicle). A vehicle/car is a tuple $k = \langle p_k, \lambda_k \rangle$, where $p_k \in V$ is the initial location of the vehicle and $\lambda_k$ is the capacity. In our setting $\lambda_k = \lambda, \forall k \in \mathcal{K}$.

We denote the set of requests by $R$, $|R| = n$ and the set of vehicles by $\mathcal{K}$, $|\mathcal{K}| = m$. Throughout the paper, we assume that $n = \lambda m$.

*Definition 4.* (Route). Given a car $k \in \mathcal{K}$, and a set of requests $R_k$ assigned to it, a route is a sequence $S_k = \langle \ell_0 = p_k, \ell_1, \ell_2, \cdots \ell_t \rangle$, starting with the origin location of car $k$, where $\ell_i \in \{s_r : r \in R_k\} \cup \{t_r : r \in R_k\}, 1 \leq i \leq t$. A route $S_k$ is **feasible** if - i) $\forall r \in R_k$, $s_r$ appears before $t_r$ in $S_k$ and ii) $|R_k| = \lambda$.

The **cost** of a route $S_k$ is defined as $\text{cost}(S_k) = \sum_{i=0}^{t-1} d(\ell_i, \ell_{i+1})$

Finally, we define the *Minimum Cost Ride Sharing* (MCRS) problem as follows.

*Definition 5.* (MCRS). Given a distance metric $d$ on a set of locations $V$, a set of requests $R$ and a set of vehicles $\mathcal{K}$, the task is to find a **feasible route** $S_k$ for each vehicle $k \in \mathcal{K}$ such that all requests in $R$ are served and the total travel distance of all vehicles, $\sum_{k \in \mathcal{K}} \text{cost}(S_k)$ is minimized.

**Computational Complexity.** The computational hardness of the MCRS problem has already been observed in previous literature and we just state it here for completeness.

Theorem 6 ([9]). *The MCRS problem is APX-hard.*

# 3 ALGORITHM FOR MCRS

## 3.1 High Level Ideas

Algorithm 1 summarizes the main steps of our algorithm which we call *Hierarchical Ride Allocation* (HRA).

The grouping phase ensures that the total length of a feasible tour to serve requests inside each group is not too large compared to the optimal solution. This requires a non-trivial *hierarchical grouping* technique which roughly works as follows. We first form sub-groups, each of size 2, such that the sum of sub-group tour costs is minimum. This is carried out using a minimum-cost matching algorithm on a suitably defined graph. In the next phase, the goal is to combine these sub-groups into larger subgroups of size 4. The idea is again to define a suitable min-cost matching problem on a graph whose vertices represent the 2-size groups formed in the

**Algorithm 1:** Hierarchical Ride Allocation

**Input:** Request set $R$. Vehicle set $\mathcal{K}$ , car capacity $\lambda$
**Output:** a set of feasible routes $S_k, \forall k \in \mathcal{K}$ such that all requests in $R$ are served

1 Run Grouping algorithm (Algorithm 2) to obtain a partition $\mathcal{P}$ of the request set $R$ such that each partition is of size $\lambda$.
2 Run Assignment and Routing algorithm (Algorithm 3) to
a ) Assign each request group in $\mathcal{P}$ to a vehicle in $\mathcal{K}$.
b ) Use Find-Route algorithm (Algorithm 4) on each vehicle to find a feasible route

first phase. However, the cost function on the edges of the graph is non-trivial now and needs to be carefully chosen such that we can bound the cost of grouping with respect to the optimal cost. This process is carried out in $\lceil \log \lambda \rceil$ iterations to obtain the final partition.

The assignment phase is a relatively straightforward min-cost matching problem on a bipartite graph where we assign each group to a vehicle minimizing the total cost of the tours.

We now describe all the steps of the HRA algorithm in more detail and prove the following central theorem.

Theorem 7. *Given a set of requests $R$ and a set of vehicles $\mathcal{K}$ with capacity $\lambda$ such that $|R| = \lambda|\mathcal{K}|$, the HRA algorithm runs in time $O(|R|^3 \log \lambda)$ and returns a set of $|\mathcal{K}|$ feasible routes serving all requests in $R$ such that the total travel distance is at most $O(\sqrt{\lambda})$ times that of an optimal solution.*

In order to formally describe the algorithm, we need some more notations. Recall that we are given a set of requests $R$, a set of vehicles $\mathcal{K}$ embedded in a metric space $(V, d)$. It is a standard fact in routing literature (see for example [12, 20]) that given a subset of points $X \in V$, the *minimum spanning tree* (MST) cost of $X$ serves as a reasonably good lower bound for the optimal travelling salesman tour on $X$. We use this heavily in our algorithm and define the following useful notations.

*Definition 8.* (Minimum Spanning Trees Cost on Requests) Let $X \subseteq R$ be a subset of requests. Then $\mathsf{mst}_s(X)$ (resp. $\mathsf{mst}_t(X)$) is defined to be the cost of a minimum spanning tree over the origins (resp. destinations) of all requests in $X$. Further, define $\mathsf{mst}_{s,t}(X) = \mathsf{mst}_s(X) + \mathsf{mst}_t(X)$.

*Definition 9.* (Incremental MST Costs) Given two requests groups $X, X' \subseteq R$, define $w(X, X') = \mathsf{mst}_{s,t}(X \cup X') - \mathsf{mst}_{s,t}(X) - \mathsf{mst}_{s,t}(X')$

Informally, $w(X, X')$ can be thought of as the *incremental MST cost* of combining request groups $X$ and $X'$ together versus serving them separately.

## 3.2 Grouping phase

In order to simplify the description of our algorithm, we assume for now that $\lambda$ is a power of 2. In Section 3.4, we explain how to modify this to work for arbitrary values of $\lambda$ with the same approximation guarantee. In Algorithm 2, requests are hierarchically combined in to groups of size 2, 4, 8, ..., $\lambda$ iteratively. Line 1 is an initialization with the trivial partition $\mathcal{P}_0$ containing all requests in $R$. Lines 3-8 iterates for $\ell = 1, 2, \cdots \log \lambda$. Consider a particular iteration $\ell$. Line 4 first builds a complete graph $G_\ell$ on the basis of the partition $\mathcal{P}_{\ell-1}$ obtained in the previous iteration. Each vertex in $G_\ell$ corresponds to a group in $\mathcal{P}_{\ell-1}$. The cost of an edge between any two vertices $P, P' \in \mathcal{P}_{\ell-1}$ is defined as $w(P, P')$, the incremental MST cost of combining the groups $P$ and $P'$. Line 5-7 then computes a minimum cost perfect matching $M_\ell$ in $G_\ell$ and uses it to combine pairs of groups $P, P'$ to obtain the new partition $\mathcal{P}_\ell$. Essentially, in iteration $\ell$, we pair up groups of size $2^{\ell-1}$ to form groups of size $2^\ell$. In the last iteration, we obtain partition $\mathcal{P} = \mathcal{P}_{\log \lambda}$, where each group contains exactly $\lambda$ requests.

**Analysis of Grouping Phase.** We now prove some important properties of the grouping phase that will help us bound total cost. Let $\mathcal{P}_\ell$ denote the partition constructed at $\ell$-th iteration (Line 7 in Algorithm 2). Based on the definition of the incremental MST cost $w$, we have the following lemma by a simple telescope sum:

**Lemma 10.** *For partition $\mathcal{P}_\ell$ ($1 \le \ell \le \log \lambda$) obtained by Algorithm 2, we have $\sum_{P \in \mathcal{P}_l} \mathsf{mst}_{s,t}(P) = \sum_{i=1}^{\ell} w(M_i)$.*

We further introduce the following notations. Fix an optimal solution $\{S_1^\star, S_2^\star, ..., S_m^\star\}$ and the corresponding partition of requests $\mathcal{P}^\star = \{$the set of requests $R_k^\star$ served in route $S_k^\star : k \in [m]\}$.

**Algorithm 2:** Grouping algorithm

**Input:** Request set $R$ and car capacity $\lambda$
**Output:** a partition $\mathcal{P}$ of $R$, each group in $\mathcal{P}$ contains $\lambda$ requests

1 $\mathcal{P}_0 = \bigcup_{r \in R}\{\{r\}\}$, $\ell = 0$
2 **while** $\ell < \log \lambda$ **do**
3  $\ell \leftarrow \ell + 1$, $\mathcal{P}_\ell = \emptyset$
4  Let $G_\ell \equiv (\mathcal{P}_{\ell-1}, E)$ be a complete graph, with weight $w(P, P')$ for any edge $(P, P') \in E$, $P, P' \in \mathcal{P}_{\ell-1}$.
5  Find a minimum weight matching $M_\ell$ in $G_\ell$ with weight $w(M_\ell) = \sum_{(P,P') \in M_\ell} w(P, P')$.
6  **for** $(P, P') \in M_\ell$ **do**
7  $\mathcal{P}_\ell \leftarrow P \cup P'$
8  **end**
9 **end**
10 **return** $\mathcal{P} = \mathcal{P}_{\log \lambda}$

**Lemma 11.** *(Grouping Lemma) The total minimum spanning tree cost on the final partition $\mathcal{P}$*

$$\sum_{P \in \mathcal{P}} \mathsf{mst}_{s,t}(P) \le O(\sqrt{\lambda}) \sum_{P \in \mathcal{P}^\star} \mathsf{mst}_{s,t}(P)$$

Proof. Suppose we have the following two inequalities (which are stated later in Lemma 12 and Lemma 13):

$$w(M_1) \le (\sqrt{\lambda} + 1) \cdot \sum_{P \in \mathcal{P}^\star} \mathsf{mst}_{s,t}(P)$$

$$w(M_\ell) \le (2^{\frac{\log \lambda - \ell}{2}} + 1) \cdot \sum_{P \in \mathcal{P}^\star} \mathsf{mst}_{s,t}(P) \text{ for } \ell \ge 2$$

Then $\sum_{P \in \mathcal{P}} \mathsf{mst}_{s,t}(P) = \sum_{\ell=1}^{\log \lambda} w(M_\ell) \le O(\sqrt{\lambda}) \sum_{P \in \mathcal{P}^\star} \mathsf{mst}_{s,t}(P)$ proves the theorem. □

Now we turn to prove Lemma 12 and 13, which bound the Incremental MST costs of every iteration $\ell$. The main idea is to break the optimal partition into sub-groups of the same size as those in $\mathcal{P}_{\ell-1}$, the partition obtained in the $(\ell-1)$-th iteration of Algorithm 2. Then we charge the cost of merging groups in $\mathcal{P}_{\ell-1}$ to the MST cost of the aforementioned sub-groups from the optimal partition.

Specifically, considering the optimal partition $\mathcal{P}^\star$, we bipartition each request group $P^\star \in \mathcal{P}^\star$ iteratively until each contains two requests: Let $\mathcal{P}^\star_0 = \mathcal{P}^\star$; in round $h \in [1, \log\lambda - 1]$, we partition the request group $P^\star \in \mathcal{P}^\star_{h-1}$ into two equal-sized sub-groups, and $\mathcal{P}^\star_h$ is the resulting partition in round $h$. Specifically, if $h$ is odd, we partition the request set $P^\star$ into two subsets $P^\star_{\text{left}}$ and $P^\star_{\text{right}}$ such that $\text{mst}_s(P^\star_{\text{left}}) + \text{mst}_s(P^\star_{\text{right}}) \le \text{mst}_s(P^\star)$; if $\ell$ is even, we partition $P^\star$ into $P^\star_{\text{left}}$ and $P^\star_{\text{right}}$ such that $\text{mst}_t(P^\star_{\text{left}}) + \text{mst}_t(P^\star_{\text{right}}) \le \text{mst}_t(P^\star)$.

To see why such bipartition scheme is always feasible, just focus on one round $h$ and w.l.o.g. assume it's odd. Then we take the MST $T_s$ on the origins of the request group $P^\star$, and break it into two equal-sized sub-trees by removing one edge from $T_s$, and $P^\star_{\text{left}}$ and $P^\star_{\text{right}}$ be the corresponding request sub-groups. By construction they satisfy $\text{mst}_s(P^\star_{\text{left}}) + \text{mst}_s(P^\star_{\text{right}}) \le \text{mst}_s(P^\star)$.

Based on this partition rule, we have the following

(1) For any even $h \ge 0$,

$$\sum_{P\in\mathcal{P}^\star_{h+2}} \text{mst}_s(P) = \sum_{P\in\mathcal{P}^\star_{h+1}} \text{mst}_s(P) \le 2 \sum_{P\in\mathcal{P}^\star_h} \text{mst}_s(P),$$
$$\sum_{P\in\mathcal{P}^\star_{h+2}} \text{mst}_t(P) \le 2 \sum_{P\in\mathcal{P}^\star_{h+1}} \text{mst}_t(P) = 2 \sum_{P\in\mathcal{P}^\star_h} \text{mst}_t(P);$$

(2) For any odd $h \ge 0$,

$$\sum_{P\in\mathcal{P}^\star_{h+2}} \text{mst}_s(P) \le 2 \sum_{P\in\mathcal{P}^\star_{h+1}} \text{mst}_s(P) = 2 \sum_{P\in\mathcal{P}^\star_h} \text{mst}_s(P),$$
$$\sum_{P\in\mathcal{P}^\star_{h+2}} \text{mst}_t(P) = \sum_{P\in\mathcal{P}^\star_{h+1}} \text{mst}_t(P) \le 2 \sum_{P\in\mathcal{P}^\star_h} \text{mst}_t(P).$$

Combine the equations above we observe that: after every *two* round of bipartitions, the total MST cost on all groups is at most doubled, i.e, for every $h \ge 0$,

$$\sum_{P\in\mathcal{P}^\star_{h+2}} \text{mst}_{s,t}(P) \le 2 \sum_{P\in\mathcal{P}^\star_h} \text{mst}_{s,t}(P). \tag{1}$$

**Lemma 12.** *The Incremental MST cost in the* 1*-st iteration*

$$w(M_1) \le (\sqrt{\lambda}+1) \cdot \sum_{P\in\mathcal{P}^\star} \text{mst}_{s,t}(P)$$

Proof. To bound the Incremental MST costs in the 1-st iteration, we break optimum groups to obtain sub-groups of size 2. After $\log\lambda - 1$ times partition, each group in $\mathcal{P}^\star_{\log\lambda-1}$ contains 2 requests and we have:

$$\sum_{P\in\mathcal{P}^\star_{\log\lambda-1}} \text{mst}_{s,t}(P) \le (2^{\frac{\log\lambda-1}{2}}+1) \sum_{P\in\mathcal{P}^\star} \text{mst}_{s,t}(P).$$

Since $\mathcal{P}_1$ is the minimum weight matching of $R$ (see Algorithm 2 Line 5), we have $w(M_1) = \sum_{P\in\mathcal{P}_1} \text{mst}_{s,t}(P) \le \sum_{P\in\mathcal{P}^\star_{\log\lambda-1}} \text{mst}_{s,t}(P)$, which proves the lemma. □

**Lemma 13.** *The Incremental MST cost in $\ell$-th ($\ell \ge 2$) iteration is*

$$w(M_\ell) \le (2^{\frac{\log\lambda-\ell}{2}}+1) \sum_{P\in\mathcal{P}^\star} \text{mst}_{s,t}(P).$$

The proof of this lemma is more involved and we refer the readers to Appendix A for full details.

## 3.3 Assignment and Routing phase

In Algorithm 3, we assign request groups formed in Algorithm 2 to vehicles and find a feasible route for each vehicle starting at their respective origin locations.

**Algorithm 3:** Assignment and Routing algorithm

**Input:** Partition $\mathcal{P}$ of request set $R$, set of vehicles $\mathcal{K}$
**Output:** A set of feasible routes $S_k$ for all vehicles $k \in \mathcal{K}$

1 **for** $k \in \mathcal{K}$ **do**
2  **for** $P \in \mathcal{P}$ **do**
3  $e(k,P) = \min_{r_i\in P} d(p_k, s_i)$
4  **end**
5 **end**
6 Let $G$ be the complete bipartite graph on $\mathcal{K} \cup \mathcal{P}$ with *left* vertex-set $\mathcal{K}$, *right* vertex-set $\mathcal{P}$, and edge weights $e(k,P)$ for $k \in \mathcal{K}$ and $P \in \mathcal{P}$.
7 Find a minimum weight perfect matching $\mathcal{A}$ in $G$.
8 **for** $k \in \mathcal{K}$ **do**
9  $S_k \leftarrow$ Find-Route$(k, P)$
10 **end**
11 **return** $S_k, \forall k \in \mathcal{K}$

**Algorithm 4:** Find-Route algorithm

**Input:** Request group $P$, vehicle $k$
**Output:** A feasible route for requests in $P$ using vehicle $k$

1 $r^\star \leftarrow \arg\min_{r\in P} d(p_k, s_r)$, $\bar{r} \leftarrow \arg\min_{r\in P} d(s_r, t_r)$
2 $\rho_0 \leftarrow$ Shortest path route between $p_k, s_{r^\star}$
3 $\rho_s \leftarrow$ 2-approximate $s$-$t$ path TSP on $\bigcup_{r\in P} s_r$ starting at $s_{r^\star}$ and ending at $s_{\bar{r}}$ using MST-heurstic [20]
4 $\rho_t \leftarrow$ 2-approximate TSP tour on $\bigcup_{r\in P} t_r$ starting at $t_{\bar{r}}$ using MST-heuristic [20]
5 $S_k \leftarrow \rho_0 \cup \rho_s \cup \{s_{\bar{r}}, t_{\bar{r}}\} \cup \rho_t$
6 **return** $S_k$

In Algorithm 3, Lines 1-5 finds for each vehicle $k \in \mathcal{K}$ and each request group $P \in \mathcal{P}$, the shortest distance $e(k,P)$ for $k$ to visit an origin location of some request in $P$. Lines 6-7 then assign vehicles to groups by computing a minimum-weight bipartite matching on $\mathcal{K}$ and $\mathcal{P}$ with edge weight $e(k,P)$. Lastly, Lines 8-10 computes a feasible route for each vehicle using Algorithm 4. In Algorithm 4, Line 1 finds the request $r^\star \in P$ whose origin is closest to the vehicle at $p_k$, i.e., $r^\star \leftarrow \arg\min_{r\in P} d(p_k, s_r)$, and also the "shortest" request $\bar{r} \in P$, i.e., $\bar{r} \leftarrow \arg\min_{r\in P} d(s_r, t_r)$. Lines 2-4 constructs the tour for vehicle $k$. It travels from $p_k$ to $s_{r^\star}$ via a shortest path, then visits all the origins using a standard 2-approximate $s$-$t$-path heuristic [20] which ends at $s_{\bar{r}}$. Finally, it moves to $t_{\bar{r}}$ and visits all

destinations using another 2-approximate TSP tour. This implies that

$$\text{cost}(S_k) \le d(p_k, s_{r^\star}) + 2\text{mst}_s(P) + d(s_{\bar{r}}, t_{\bar{r}}) + 2\text{mst}_t(P).$$

**Approximation Analysis.** In the remainder of this section, we prove a bound on the total cost of the tours returned by Algorithm 3 and prove Theorem 7.

We first bound the cost of visiting first locations, $s_{r^\star}$. Recall the notations: $\{S_1^\star, S_2^\star, ..., S_m^\star\}$ is any fixed optimal solution, and $\mathcal{P}^\star = \{\text{the set of requests } R_k^\star \text{ served in route } S_k^\star : k \in [m]\}$. We have the following structural lemma that relates any feasible partition $\mathcal{P}$ with $\mathcal{P}^\star$ (the full proof can be found in Appendix A).

**Lemma 14.** *Suppose $G \equiv (\mathcal{P}, \mathcal{P}^\star, E)$ is a bipartite, $\lambda$-regular, multi-graph, where for each request $r \in R$ there is an edge between vertex $P \in \mathcal{P}$ and $R_k^\star \in \mathcal{P}^\star$ with weight $f(r)$, whenever $r \in P \cap R_k^\star$. We can find $\lambda$ disjoint perfect matchings in graph $G$ with total weight $\sum_{r\in R} f(r)$.*

Let us set $f(r_i) = d(p_k, s_i)$ in Lemma 14 for each request $r_i \in R_k^\star$ with $R_k^\star \in \mathcal{P}^\star$. Then by averaging, the lemma implies that there exists one perfect matching $M$ in $G \equiv (\mathcal{P}, \mathcal{P}^\star, E)$ with cost no more than $\frac{1}{\lambda}\sum_{R_k^\star\in\mathcal{P}^\star}\sum_{r_i\in R_k^\star} d(p_k, s_i)$. By assigning group $P$ to vehicle $k$ for each $(P, R_k^\star) \in M$, we get a perfect matching $\mathcal{A}'$ in $G \equiv (\mathcal{K} \cup \mathcal{P}, e)$ with $e(\mathcal{A}') \le \frac{1}{\lambda}\sum_{R_k^\star\in\mathcal{P}^\star}\sum_{r_i\in R_k^\star} d(p_k, s_i)$. Now recall Algorithm 3 finds a minimum weight matching $\mathcal{A}$ in $G \equiv (\mathcal{K} \cup \mathcal{P}, e)$, we have the following inequality on the cost of $\mathcal{A}$.

$$e(\mathcal{A}) = \sum_{k\in\mathcal{K}} d(p_k, s_{r^\star}) \le e(\mathcal{A}') \le \frac{1}{\lambda}\sum_{R_k^\star\in\mathcal{P}^\star}\sum_{r_i\in R_k^\star} d(p_k, s_i). \quad (2)$$

Now we apply Lemma 14 again but this time let $f(r_i) = d(s_i, t_i)$. We know that there is a perfect matching in $G \equiv (\mathcal{P}, \mathcal{P}^\star, E)$ with cost smaller than $\frac{1}{\lambda}\sum_{R_k^\star\in\mathcal{P}^\star}\sum_{r_i\in R_k^\star} d(s_i, t_i)$, and thus

$$\sum_{P\in\mathcal{P}} d(s_{\bar{r}}, t_{\bar{r}}) = \sum_{P\in\mathcal{P}} \min_{r_i\in P} d(s_i, t_i) \le \frac{1}{\lambda}\sum_{R_k^\star\in\mathcal{P}^\star}\sum_{r_i\in R_k^\star} d(s_i, t_i). \quad (3)$$

We are now ready to complete the proof of Theorem 7. Using the Grouping Lemma (Lemma 11) and inequality (2) and (3), we have:

$$\sum_{k\in\mathcal{K}} \text{cost}(S_k) \le \sum_{(k,P)\in\mathcal{A}} \left(d(p_k, s_{r^\star}) + 2\text{mst}_{s,t}(P) + d(s_{\bar{r}}, t_{\bar{r}})\right) \quad (4)$$

$$\le O(\sqrt{\lambda}) \sum_{P\in\mathcal{P}^\star} \text{mst}_{s,t}(P) + \frac{1}{\lambda}\sum_{R_k^\star\in\mathcal{P}^\star}\sum_{r_i\in R_k^\star} (d(p_k, s_i) + d(s_i, t_i)) \quad (5)$$

$$\le O(\sqrt{\lambda}) \sum_{k\in\mathcal{K}} \text{cost}(S_k^\star) \quad (6)$$

Note that for any request $r_i \in R_k^\star$ with $R_k^\star \in \mathcal{P}^\star$, there is $\text{cost}(S_k^\star) \ge (1/2)\text{mst}_{s,t}(R_k^\star)$ and $\text{cost}(S_k^\star) \ge d(p_k, s_i) + d(s_i, t_i)$, which finishes the proof of inequality (6).

**Runtime Analysis.** In the Grouping phase, we construct $\frac{n}{2^\ell}$ minimum spanning trees in each iteration $\ell$, each of them can be constructed in time $O(2^{2\ell}\log(2^\ell))$ [15]; and we compute $\log\lambda$ minimum-cost perfect matchings in Grouping phase, each of them can be found in time $O(n^3)$ [8]; Thus, the running time of Grouping phase is $O(n^3\log\lambda)$. In the Assignment and Routing phase, we found minimum perfect matching in time $O(m^3)$. In total, the running time is $O(n^3\log\lambda)$.

## 3.4 Algorithm for general capacities

In this section, we briefly mention how to modify HRA in order to handle values of $\lambda$ that are (possibly) not exact powers of 2. We just need to modify the Grouping algorithm (Algorithm 2). We will have a *forward* and *backward* phase. First observe that any positive integer $\lambda$ can be written as $\lambda = \sum_{\ell=0}^{\lfloor\log\lambda\rfloor} b_i \cdot 2^\ell$, where $b_\ell \in \{0, 1\}$. Recall in iteration $\ell$ of Algorithm 2, we compute a minimum cost matching that combines groups of size $2^{\ell-1}$ into groups of size $2^\ell$. In the modified algorithm, if $b_\ell = 0$, we do the same as before; otherwise, if $b_\ell = 1$, we find a min-cost matching that matches *all but discards m* groups of size $2^{\ell-1}$. After $\lfloor\log\lambda\rfloor$ iterations, we would have formed groups of size $2^{\lfloor\log\lambda\rfloor}$. However, we still need to combine the groups that we potentially discarded in previous iterations. To this end, we perform a backward phase - we iterate for $\ell = \lfloor\log\lambda\rfloor$ down to 0. At iteration $\ell$, we set up a minimum cost bipartite matching with the discarded groups from iteration $\ell$ on one side and the already formed groups on the other side. On the basis of this matching, we update each group with $2^\ell$ more requests. We remark that all the proofs from the previous section goes through with some minor modifications and the modified HRA algorithm is still an $O(\sqrt{\lambda})$-approximation.

# 4 EXPERIMENTS

## 4.1 Experiment Setup

**Metrics** In our experiments, we compare the performance of our algorithm (Hra) with various baselines. The relative performance of the algorithms is measured across four aspects: total travel distance, makespan, total latency, and running time. Aside from total travel distance, *makespan* and *total latency* are two other commonly used objectives in vehicle routing problems. Makespan is defined as the maximum travel distance among all vehicles, while total latency is the sum of serving time of each request (Assume all vehicles leave their depot at time 0, the serving time of a request is the time when it is delivered to its destination). Recall all vehicles are running at unit speed, thus the "time" here is equivalent to the distance a vehicle has travelled.

**Baselines.** We use the following algorithms as baselines. Since there is no algorithm that solves exactly the same problem as ours (as far as we know), we have to modify these baselines to make them work with our setting.

(1) Greedy. This is a simple greedy method which can be thought of as the naive version of our main algorithm. Instead of the hierarchical grouping scheme, we aggregate requests by repeatedly solving capacity-2 sub-instances and stack them together: first run Algorithm 2 *for one iteration* to get all requests paired; then we greedily assign these pairs by repeatedly computing bipartite matchings between the vehicles and the *unmatched* request pairs. The cost for matching a request pair to a vehicle is the incremental distance that the vehicle would need to travel to serve the pair along with the already-matched requests. If $\lambda m$ is odd, at the last iteration there will be $m$ requests left, which will be assigned by another

min-cost matching with the vehicles. Lastly, we run Algorithm 4 to compute a route for each vehicle.

We remark that the Greedy algorithm runs in $O(\lambda n^3)$ time, and it's easy to show a $\Omega(\lambda)$ lower bound for its approximation ratio.

(2) pruneGdp[18]. This is Algorithm 5 in [18]. It was designed for the multi-vehicle dial-a-ride problem and does not directly work with our problem. (see Section 1 for a comparison of the two problems). The key difference is that a vehicle can be reused multiple times and could serve more than $\lambda$ requests, while in our setting a vehicle can only be used once. We modify their algorithm by forcing each vehicle to serve no more than $\lambda$ requests.

(3) Lmd[22]. This is Algorithm 1 in [22]. Like pruneGdp, it was designed for the dial-a-ride problem and we modify it to make sure no vehicle serves more than $\lambda$ requests.

**Implementation** We implement our algorithm and Greedy using Python, and use the C++ implementation provided by [22] for pruneGdp and Lmd. The experiments are conducted on a machine with 6 Intel 2.2GHZ cores and 16GB RAM.

**Synthetic Datasets.** We use a similar method as in [2] to generate the synthetic datasets in the Euclidean plane. Specifically, we first generate some $K$ centers uniformly at random from a $4000 \times 4000$ box on the Euclidean plane. These centers will serve as the means of $K$ Gaussian distribution with covariance $\Sigma$. Then for various request numbers $n$ and vehicle capacity $\lambda$, we sample from the mixture of $K$ Gaussians $m = n/\lambda$ vehicle locations, $n$ request origins, and another $n$ corresponding destinations. The detailed parameter setting is summarized in Table 1(a), where numbers in bold are fixed when varying other parameters: e.g., we fix $n = 5040$ when varying $\lambda$. For each parameter setting, the experiment is repeated 25 times and the average performance is reported.

**(a) Synthetic datasets.**

| | |
|---|---|
| $n$: #requests | 840, 2520, **5040**, 7560, 10080 |
| $\lambda$: vehicle capacity | 2, **4**, 5, 6, 8 |
| $K$: #clusters of the GMM | 5, **10**, 20, 50 |
| $\sigma$: covariance of the GMM | $10I$, $50I$, **100**$I$, $200I$ |

**(b) Realworld datasets.**

| Dataset | $n$: #requests | $\lambda$: vehicle capacity |
|---|---|---|
| NYC | 600, 1080, **2640**, 5040, 7560 | 2, **4**, 5, 6, 8 |
| SFO | 480, 840, **1680**, 3000, 4560 | 2, **4**, 5, 6, 8 |

**Table 1: Parameter settings for the datasets. A parameter is fixed to the number in bold when varying other parameters.**

**Real-world Datasets.** We test our algorithm on real-world taxi records for New York City (NYC) and San Francisco (SFO). For NYC we use the NYC TLC Trip Record Data[17], which contains taxi transportation records through year 2009 to 2020. In our experiment we pick the subset of records from April 2016. Each record consists of the pick-up and drop-off time, the origin and destination locations, and trip distances. The locations are specified using GPS coordinates. For SFO, we use the Cab Spotting Data[16] that contains trace records of approximately 500 taxis collected over 30 days in the San Francisco Bay Area. We use the shortest-path distance defined by the Open Street Maps[14], which contains road network map of both cities: the map for NYC has 1,014,391 vertices and 1,140,405 edges, while that for SFO has 210,937 vertices and 229,581 edges. (Note our algorithm's complexity is independent of the size of the underlying graph, because what we only need is the pairwise distance between car/request locations, which can be pre-computed.)

The parameter setting is summarized in Table 1(b). To vary the number of requests, we pick time windows with length ranging from $1 \sim 15$ minutes, and run experiments for the subset of data within each window. Also, there is no information for initial vehicle locations in the dataset, therefore we follow the strategy used in [22] and generate initial vehicle locations by sampling vertices uniformly from the Open Street Maps.

## 4.2 Experimental Results

*4.2.1 Results on synthetic datasets.* Figure 1 shows the experimental result on synthetic datasets. Each column shows one objective under various parameter settings (from left to right): total travel distance, makespan, total latency, and running time. Each row corresponds to the result of varying one parameter (from top to bottom): number of requests ($n$), vehicle capacity ($\lambda$), GMM centers ($K$), and GMM covariance ($\Sigma$).

**Effect of varying number of requests.** Our algorithm (Hra) consistently performs (among) the best on all three objectives. The gap is especially large compared with Lmd and pruneGdp: it obtains total travel distance (1st column) 80%-less than Lmd and 50%-less than pruneGdp, and a similar margin for the total latency objective (3rd column). For the makespan objective, Hra is still significantly ($\sim 50\%$) better than Lmd, and outperforms pruneGdp by a small margin. Regarding running time, pruneGdp is always much faster than other algorithms, while Hra is comparable with Lmd.

The interesting case is the Greedy baseline: it achieves almost the same performance as Hra on all three objectives, albeit with a worse running time (Note we don't have results for Greedy when $n > 8000$ because it takes too long to finish, but we expect it to perform similar as Hra). Although in theory Greedy could perform worse than Hra ($\Omega(\lambda)$ versus $\tilde{O}(\sqrt{\lambda})$), the difference may not be noticeable when $\lambda$ is small.

**Effect of varying capacity.** The effect of varying capacity $\lambda$ is more subtle, though. Since we work with the setting $n = \lambda m$, a larger $\lambda$ implies less vehicles. Hence, the total travel distance decreases for all the algorithms. However, some of the vehicles might now travel longer distances which results in an increase in the makespan and total latency.

**Effects of other parameters.** Overall, varying the parameters, number of cluster centers ($K$) or variance of the Gaussian ($\Sigma$), does not alter the relative performance of the algorithms being compared (except for running time). Increasing the number of requests (1st row) unsurprisingly leads to larger total travel distance or latency, while the makespan is less affected as it's more related to the spatial distribution of requests, which is reflected clearly when increasing the variance of sample data distribution.

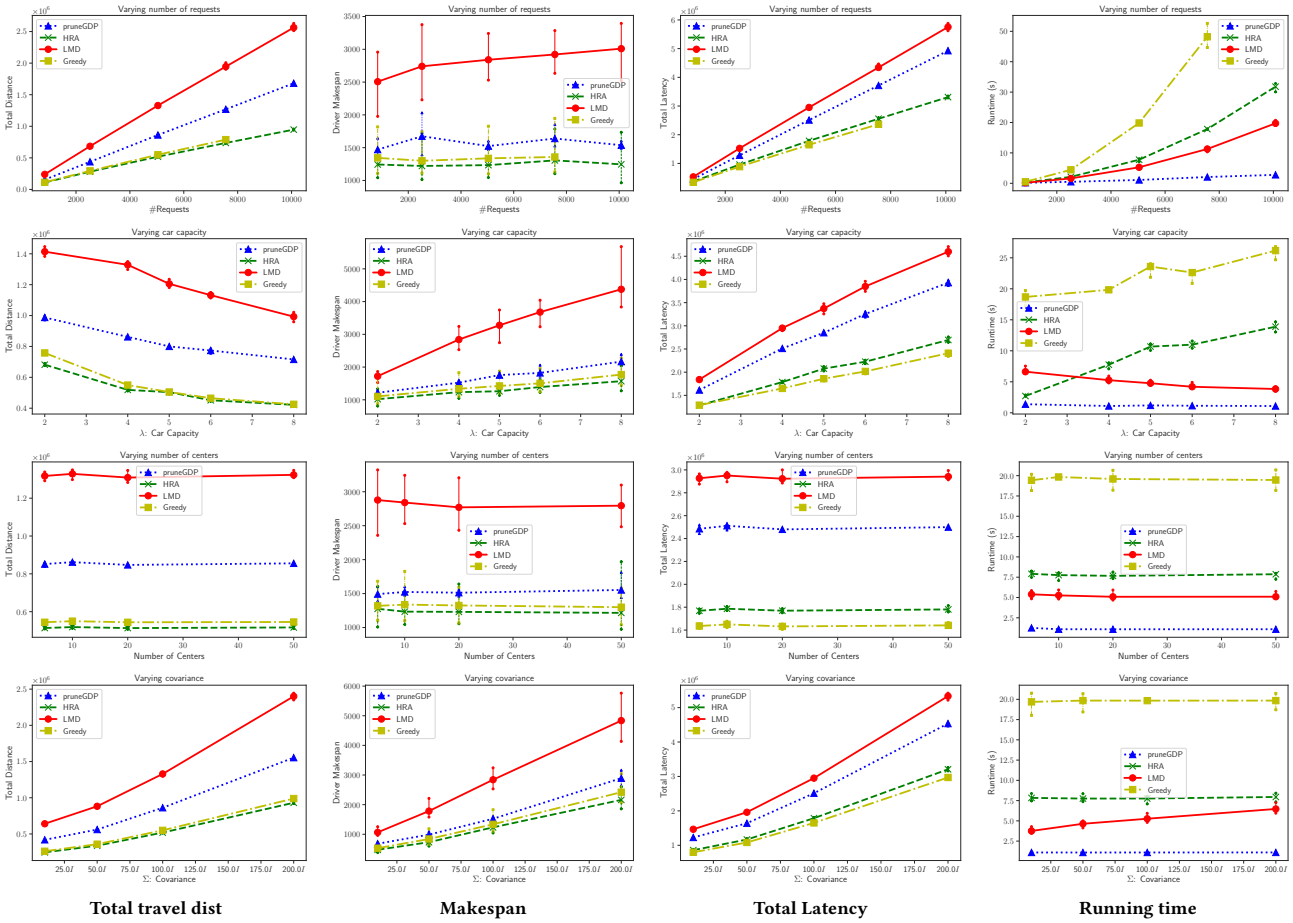


**Figure 1: Results on the synthetic data set. From top to bottom: (1) Varying request number $n$ (with vehicle capacity fixed $\lambda = 4$); (2) Varying $\lambda$ with fixed $n = 5040$; (3) Varying number of centers $K$ of the GMM, with fixed $n = 5040$ and $\lambda = 4$; (4) Varying covariance $\Sigma$ of the GMM, with fixed $n = 5040$, $\lambda = 4$, and $K = 10$.**

*4.2.2 Results on real-world datasets.* Figure 2 and 3 show the results on real-world taxi transportation datasets.

Just like the situation in synthetic datasets, on both datasets our algorithm HRA, along with GREEDY, achieves the best performance on all three objectives. But this time the advantage of HRA is smaller: when the capacity is fixed (the top rows of Figure 2 and 3), HRA achieves total travel distance 20% less than LMD and 30% less than PRUNEGDP, and a 10%-20% smaller total latency. The gap between LMD and HRA is even larger on the makespan objective, with LMD giving almost twice as large makespan; while PRUNEGDP shows a smaller margin, but still gives 20%-30% larger makespan. If we fix the number of requests and change the capacity (the bottom rows of Figure 2 and 3), HRA (and GREEDY) still consistently outperforms the other two baselines by a significant margin.

One notable difference between the results and those on the synthetic datasets is the running time. PRUNEGDP is still the fastest, and significantly faster than all other three algorithms; while GREEDY now takes comparable time with LMD and HRA. The main reason is because we are using the shortest-path distance, and the number of shortest-path queries in PRUNEGDP is significantly smaller than other three algorithms. PRUNEGDP makes roughly $O(n)$ many queries, while the other three require $\Omega(n^2)$ queries. Because the underlying graph is huge, the distance query is expensive and dominates the running time.

**Handling dynamic requests.** Although our algorithm has been designed for static input, it can be deployed effectively in a dynamic request scenario using the standard batching technique [25]. The idea is to collect all requests coming inside a pre-determined time window and process them together. For example, in the NY dataset, during the most busy hours, the number of requests in windows of size 15 minutes is approximately 7500. Our algorithm can process these requests in less than 100 seconds (See Figure 2) which is significantly small compared to the size of the time-window.

*4.2.3 Summary.*

- In terms of total travel distance, our algorithm HRA gives notably better solutions than LMD and PRUNEGDP. This is

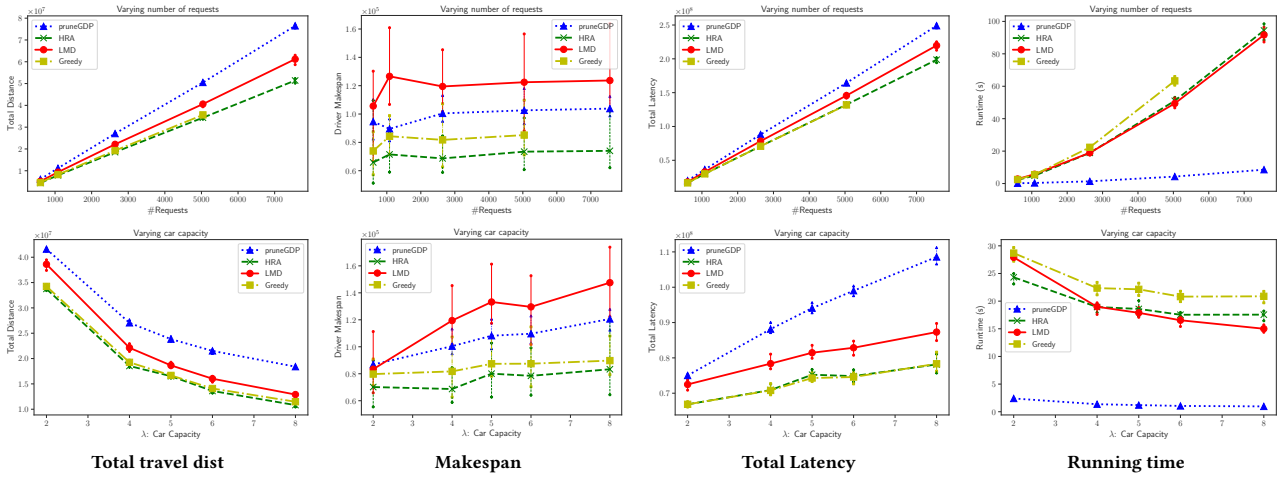


**Figure 2: Results on the NYC dataset. The first row is the result of varying $n$, and the second row is for varying $\lambda$.**

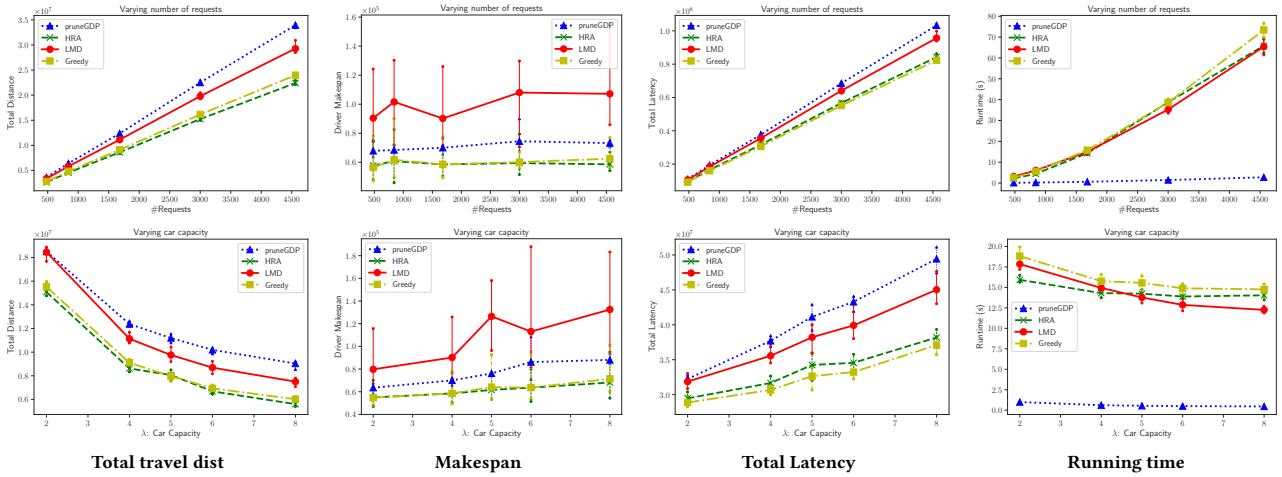


**Figure 3: Results on the SFO dataset. The first row is the result of varying $n$, and the second row is for varying $\lambda$.**

expected, as HRA directly optimize the objective. But HRA also shows good robustness in the sense that it also performs well on the makespan and total latency objectives.

- In terms of running time, HRA is slower than the baseline pruneGDP, but still comparable with LMD.
- The GREEDY method, which can be viewed as a naive version of HRA and is strictly worse in theory, achieves comparable experimental performance as HRA. We believe this offers another option for practitioners, as GREEDY is conceptually much simpler and easier to implement.

## 5 CONCLUSION

In this paper, we give the first algorithm, HRA, for the minimum total distance ride-sharing problem with vehicles of any arbitrary vehicle capacity $\lambda \geq 2$. We prove an approximation guarantee of $O(\sqrt{\lambda})$ for HRA. We also give a greedy heuristic that has comparable performance to HRA in practice. Extensive experiments on both synthetic and real-world datasets testify to the fact that our algorithms have up to 40% performance improvement compared to some of the state-of-the-art ride-sharing algorithms. Finally, we witness that in practice, our algorithms either match or outperform existing baselines even for the complementary objectives of makespan and total latency of individual riders.

## ACKNOWLEDGMENTS

Kelin Luo has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 754462.

## REFERENCES


[1] Javier Alonso-Mora, Samitha Samaranayake, Alex Wallar, Emilio Frazzoli, and Daniela Rus. 2017. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proceedings of the National Academy of Sciences* 114, 3 (2017), 462–467.

[2] Xiaohui Bei and Shengyu Zhang. 2018. Algorithms for Trip-Vehicle Assignment in Ride-Sharing. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 3–9. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16583

[3] Blablacar. 2021. Blablacar. https://www.blablacar.com/

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

[5] Moses Charikar and Balaji Raghavachari. 1998. The finite capacity dial-a-ride problem. In *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*. IEEE, 458–467.

[6] Regina R Clewlow and Gouri S Mishra. 2017. Disruptive transportation: The adoption, utilization, and impacts of ride-hailing in the United States. (2017).

[7] Nicolas Coulombel, Virginie Boutueil, Liu Liu, Vincent Viguie, and Biao Yin. 2019. Substantial rebound effects in urban ridesharing: Simulating travel decisions in Paris, France. *Transportation Research Part D: Transport and Environment* 71 (2019), 110–126.

[8] 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*. 434–443.

[9] Dries Goossens, Sergey Polyakovskiy, Frits CR Spieksma, and Gerhard J Woeginger. 2012. Between a rock and a hard place: the two-to-one assignment problem. *Mathematical methods of operations research* 76, 2 (2012), 223–237.

[10] 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.

[11] Jagan Jacob and Ricky Roet-Green. 2021. Ride solo or pool: Designing price-service menus for a ride-sharing platform. *European Journal of Operational Research* (2021).

[12] Eugene L Lawler. 1985. The traveling salesman problem: a guided tour of combinatorial optimization. *Wiley-Interscience Series in Discrete Mathematics* (1985).

[13] Kelin Luo and Frits CR Spieksma. 2020. Approximation algorithms for car-sharing problems. In *International Computing and Combinatorics Conference*. Springer, 262–273.

[14] Open Street Map. 2011. Open Street Map. https://download.bbbike.org/

[15] Seth Pettie and Vijaya Ramachandran. 2000. An optimal minimum spanning tree algorithm. In *International Colloquium on Automata, Languages, and Programming*. Springer, 49–60.

[16] Michal Piorkowski, Natasa Sarafijanovic-Djukic, and Matthias Grossglauser. 2009. CRAWDAD dataset epfl/mobility (v. 2009-02-24). Downloaded from https://crawdad.org/epfl/mobility/20090224/cab. https://doi.org/10.15783/C7J010 traceset: cab.

[17] TLC Trip Record Data TLC. 2016. New York City TLC trip record data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[18] Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, Jieping Ye, and Ke Xu. 2018. A Unified Approach to Route Planning for Shared Mobility. *Proc. VLDB Endow.* 11, 11 (2018), 1633–1646. https://doi.org/10.14778/3236187.3236211

[19] Transvision. 2021. Transvision. https://www.transvision.nl/

[20] René van Bevern and Viktoriia A Slugina. 2020. A historical note on the 3/2-approximation algorithm for the metric traveling salesman problem. *Historia Mathematica* 53 (2020), 118–127.

[21] Biying Yu, Ye Ma, Meimei Xue, Baojun Tang, Bin Wang, Jinyue Yan, and Yi-Ming Wei. 2017. Environmental benefits from ridesharing: A case of Beijing. *Applied energy* 191 (2017), 141–152.

[22] Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2019. Last-Mile Delivery Made Practical: An Efficient Route Planning Framework with Theoretical Guarantees. *Proc. VLDB Endow.* 13, 3 (2019), 320–333. https://doi.org/10.14778/3368289.3368297

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

[24] Libin Zheng, Lei Chen, and Jieping Ye. 2018. Order Dispatch in Price-Aware Ridesharing. *Proc. VLDB Endow.* 11, 8 (April 2018), 853–865.

[25] Libin Zheng, Peng Cheng, and Lei Chen. 2019. Auction-Based Order Dispatch and Pricing in Ridesharing. In *ICDE*. IEEE, 1034–1045.


## A MISSING PROOFS

**Lemma 13.** *The Incremental MST cost in $\ell$-th ($\ell \geq 2$) iteration is*

$$w(M_\ell) \leq (2^{\frac{\log\lambda-\ell}{2}}+1)\sum_{P\in\mathcal{P}^\star}\mathrm{mst}_{s,t}(P).$$

Proof. To bound the Incremental MST costs in the $\ell$-st iteration, we break optimum groups to obtain sub-groups of size $2^\ell$. By inequality (1), we have

$$\sum_{P\in\mathcal{P}^\star_\ell}\mathrm{mst}_{s,t}(P) \leq (2^{\frac{\ell}{2}}+1)\sum_{P\in\mathcal{P}^\star}\mathrm{mst}_{s,t}(P).$$

Therefore if we can prove $w(M_\ell) \leq \sum_{P\in\mathcal{P}^\star_{\log\lambda-\ell}}\mathrm{mst}_{s,t}(P)$, then

$$w(M_\ell) \leq (2^{\frac{\log\lambda-\ell}{2}}+1)\sum_{P\in\mathcal{P}^\star}\mathrm{mst}_{s,t}(P).$$

Now we show how to do this. Recall the partitions of $\mathcal{P}^\star$ constructed in Lemma 12, i,e., $\{\mathcal{P}^\star_1, \mathcal{P}^\star_2, ..., \mathcal{P}^\star_{\log\lambda-1}\}$, where each request group in partition $\mathcal{P}^\star_{\log\lambda-\ell}$ contains $2^\ell$ requests for all $\ell \geq 0$. Define a bipartite, $2^{\ell-1}$-regular multi-graph $(\mathcal{P}_{\ell-1}, \mathcal{P}^\star_{\log\lambda-(\ell-1)}, E)$ as follows: the vertices of the graph are the groups from $\mathcal{P}_{\ell-1}$ or $\mathcal{P}^\star_{\log\lambda-(\ell-1)}$, and for each request $r$ we add an edge $e$ between vertex $P \in \mathcal{P}_{\ell-1}$ and $P^\star \in \mathcal{P}^\star_{\log\lambda-(\ell-1)}$ if $r \in P \cap P^\star$. Note that $|E| = |R| = 2^{\ell-1}\cdot|\mathcal{P}_{\ell-1}|$. According to Hall's marriage theorem, we can find a perfect matching $\mathcal{M}$ among $\mathcal{P}_{\ell-1}$ and $\mathcal{P}^\star_{\log\lambda-(\ell-1)}$.

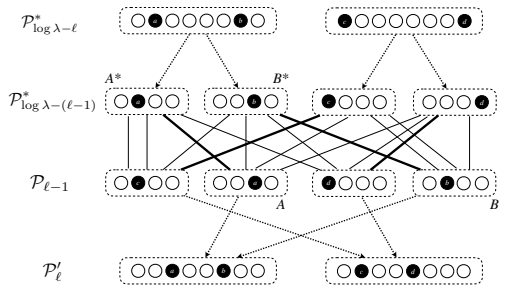


**Figure 4: Visualization of $\mathcal{P}'_\ell$ in Lemma 13. Find a partition $\mathcal{P}'_\ell$: we first find a matching $M$ between $\mathcal{P}_{\ell-1}$ and $\mathcal{P}^\star_{\log\lambda-(\ell-1)}$ (see bold black lines); then find partition $\mathcal{P}'_\ell$ based on $M$ and $\mathcal{P}^\star_{\log\lambda-\ell}$ (see dotted lines)**

This matching $\mathcal{M}$ naturally induces a merging scheme for the partition $\mathcal{P}_{\ell-1}$. Take two groups $A^\star$ and $B^\star$ in $\mathcal{P}^\star_{\log\lambda-(\ell-1)}$ that are obtained by breaking a same larger group in $\mathcal{P}^\star_{\log\lambda-\ell'}$. Then consider their matched groups (under $\mathcal{M}$) in $\mathcal{P}_{\ell-1}$, denoted as $A$ and $B$ respectively: we merge $A, B$ to get a larger group. This procedure gives us a partition $\mathcal{P}'_\ell$, each group of which is obtained by merging two groups of $\mathcal{P}_{\ell-1}$

Now we claim the Incremental MST cost of getting $\mathcal{P}'_\ell$ is bounded by $\sum_{P\in\mathcal{P}^\star_{\log\lambda-\ell}}\mathrm{mst}_{s,t}(P)$. Then since $\mathcal{P}_\ell$ is obtained by computing a *min-cost* matching, its cost increase (i.e., $w(M_\ell)$) must also be less than $\sum_{P\in\mathcal{P}^\star_{\log\lambda-\ell}}\mathrm{mst}_{s,t}(P)$.

Firstly, by the definition of $\mathcal{M}$ we know both $A\cap A^\star$ and $B\cap B^\star$ are non-empty. Thus, for $P^\star = A^\star \cup B^\star \in \mathcal{P}^\star_{\log\lambda-\ell}$ we have

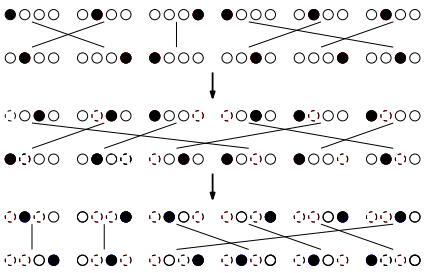

**Figure 5: An example with $\lambda = 4$ and $|\mathcal{P}| = 6$: when $j = 1$, find a perfect matching in $(\mathcal{P}_1, \mathcal{P}_1^*, E_1)$, delete the common requests in each matched request groups and obtain graph $(\mathcal{P}_j, \mathcal{P}_j^*, E_j)$ for $j = 2$; Similarly, find the rest three perfect matchings.**

$\min_{r_i \in A, r_j \in B} d(s_i, s_j) + \min_{r_i \in A, r_j \in B} d(t_i, t_j) \le \text{mst}_s(P^\star) + \text{mst}_t(P^\star)$. Now the crucial observation is $w(A, B) \le \min_{r_i \in A, r_j \in B} d(s_i, s_j) + \min_{r_i \in A, r_j \in B} d(t_i, t_j)$: to see this, recall the definition of $w(A, B)$,

$$w(A, B) = (\text{mst}_s(A \cup B) - \text{mst}_s(A) - \text{mst}_s(B)) + (\text{mst}_t(A \cup B) - \text{mst}_t(A) - \text{mst}_t(B)).$$

Note that $\text{mst}_s(A \cup B) - \text{mst}_s(A) - \text{mst}_s(B) \le \min_{r_i \in A, r_j \in B} d(s_i, s_j)$: Let's denote the MSTs on the origins of A and B as $\text{MST}_s(A)$ and $\text{MST}_s(B)$, and let $(r_a, r_b) = \arg\min_{r_i \in A, r_j \in B} d(s_i, s_j)$ be the two requests whose origins are the closest. Then adding one edge $(s_a, s_b)$ connecting $\text{MST}_s(A)$ and $\text{MST}_s(B)$ results in a spanning tree $T_s'$ on the origins of $A \cup B$, and this $T_s'$ has incremental cost $d(s_a, s_b)$; But by definition $\text{cost}(T') \le \text{mst}_s(A \cup B)$, which gives the claimed inequality. Similarly, $\text{mst}_t(A \cup B) - \text{mst}_t(A) - \text{mst}_t(B) \le \min_{r_i \in A, r_j \in B} d(t_i, t_j)$. Therefore we have

$$w(A, B) \le \min_{r_i \in A, r_j \in B} d(s_i, s_j) + \min_{r_i \in A, r_j \in B} d(t_i, t_j)$$

and thus $\sum_{P = A \cup B \in \mathcal{P}_\ell'} w(A, B) \le \sum_{P \in \mathcal{P}^\star_{\log \lambda - \ell}} \text{mst}_{s,t}(P)$. This concludes our proof.

□

**Lemma 14.** *Suppose $G \equiv (\mathcal{P}, \mathcal{P}^\star, E)$ is a bipartite, $\lambda$-regular, multigraph, where for each request $r \in R$ there is an edge between vertex $P \in \mathcal{P}$ and $R_k^\star \in \mathcal{P}^\star$ with weight $f(r)$, whenever $r \in P \cap R_k^\star$. We can find $\lambda$ disjoint perfect matchings in graph $G$ with total weight $\sum_{r \in R} f(r)$.*

Proof. For any two perfect matchings $M_i, M_j$ in graph $G \equiv (\mathcal{P}, \mathcal{P}^*, E)$, $M_i$ and $M_j$ are *disjoint* if and only if $M_i \cap M_j = \emptyset$. Suppose there are $\lambda$ perfect matching $\mathcal{M}$ such that any two perfect matchings $M_i, M_j \in \mathcal{M}$ are disjoint, we know that $\mathcal{M}$ contains all edges of $E$, then the sum of the weight of the $\lambda$ perfect matchings is:

$$\sum_{M_i \in \mathcal{M}} \sum_{e(r) \in M_i} f(r) = \sum_{r \in R} f(r)$$

We construct $\lambda$ disjoint perfect matchings in graph $G \equiv (\mathcal{P}, \mathcal{P}^*, E)$ by the following steps (see Figure 5 as an example):

Step (1) Initialization $j = 1$, graph $G_j \equiv (\mathcal{P}_j, \mathcal{P}_j^*, E_j) \equiv (\mathcal{P}, \mathcal{P}^*, E)$;

Step (2) Find a perfect matching $M_j$ in graph $(\mathcal{P}_j, \mathcal{P}_j^*, E_j)$, then delete the edges of $M_j$ (also the corresponding *requests* of these edges from its request groups) from graph $(\mathcal{P}_j, \mathcal{P}_j^*, E_j)$, and obtain a bipartite graph $G_{j+1} \equiv (\mathcal{P}_{j+1}, \mathcal{P}_{j+1}^*, E_{j+1})$;

Step (3) Repeat Step (2) until $j = \lambda$.

We claim that there always exists a perfect matching in Step (2). Since we delete all edges of $\bigcup_{i \le j} M_i$ from the graph $G \equiv (\mathcal{P}, \mathcal{P}^*, E)$ before create matching $M_{j+1}$, we know that all edges $E = \bigcup_{i=1}^{\lambda} M_i$ and then $\sum_{i=1}^{\lambda} \sum_{e(r) \in M_i} f(r) = \sum_{r \in R} f(r)$ as the claim holds.

We prove the claim by repeatedly applying Hall's marriage theorem, which claims that for any bipartite graph $G = (X \cup Y, E)$, there is an $X$-perfect matching if and only if $|W| \le |N_G(W)|$ for every subset $W$ of $X$ where $N_G(W)$ denote the neighborhood of $W$ in $G$. We prove the following by induction on $j$: $\forall j \in \{0, 1, 2..., \lambda - 1\}$ and graph $(\mathcal{P}_{j+1}, \mathcal{P}_{j+1}^*, E_{j+1})$, there is $|P| = \lambda - j$ for all $P \in \mathcal{P}_{j+1}$ or $P \in \mathcal{P}_{j+1}^*$, and $\bigcup_{P \in \mathcal{P}_{j+1}} P = \bigcup_{P \in \mathcal{P}_{j+1}^*} P$; Furthermore, there is a perfect matching in graph $(\mathcal{P}_{j+1}, \mathcal{P}_{j+1}^*, E_{j+1})$.

**Base Case:** $j = 0$. Obviously $\bigcup_{P \in \mathcal{P}_1} P = \bigcup_{P \in \mathcal{P}_1^*} P^* = R$, $|P| = \lambda$ for all $P \in \mathcal{P}_1$ or $P \in \mathcal{P}_1^*$. Since the number of requests in $W$ is $\lambda \cdot |W|$ for every subset $W$ of $\mathcal{P}_1^*$, $|N_G(W)| \ge \frac{\lambda \cdot |W|}{\lambda} = |W|$ (each item in $N_G(W)$ contains exactly $\lambda$ requests) where $N_G(W)$ denote the neighborhood of $W$ in $(\mathcal{P}_1, \mathcal{P}_1^*, E_1)$. By Hall's theorem, there is a perfect matching in $G_1 \equiv (\mathcal{P}_1, \mathcal{P}_1^*, E_1)$.

**Induction:** Suppose the claim holds in graph $(\mathcal{P}_j, \mathcal{P}_j^*, E_j)$. After deleting the edges of $M_j$ (also the requests on edges $M_j$ from its corresponding request groups $P \in \mathcal{P}_j$ and $P^* \in \mathcal{P}_j^*$) from graph $G_j \equiv (\mathcal{P}_j, \mathcal{P}_j^*, E_j)$, we obtain a bipartite graph $G_{j+1} \equiv (\mathcal{P}_{j+1}, \mathcal{P}_{j+1}^*, E_{j+1})$ and $|P| = \lambda - j$ for all $P \in \mathcal{P}_{j+1}$ or $P \in \mathcal{P}_{j+1}^*$ because $|P| = \lambda - j + 1$ for all $P \in \mathcal{P}_j$ or $P \in \mathcal{P}_j^*$ by the induction hypothesis. Also we have $\bigcup_{P \in \mathcal{P}_{j+1}} P = \bigcup_{P \in \mathcal{P}_{j+1}^*} P$. Since the number of requests in $W$ is $(\lambda - j) \cdot |W|$ for every subset $W$ of $\mathcal{P}_{j+1}^*$, $|N_G(W)| \ge \frac{(\lambda - j) \cdot |W|}{\lambda - j} = |W|$ (each item in $N_G(W)$ contains exactly $\lambda - j$ requests) where $N_G(W)$ denote the neighborhood of $W$ in $G_{j+1} \equiv (\mathcal{P}_{j+1}, \mathcal{P}_{j+1}^*, E_{j+1})$. Again by Hall's theorem, there is a perfect matching in $G_{j+1} \equiv (\mathcal{P}_{j+1}, \mathcal{P}_{j+1}^*, E_{j+1})$. □