Journal of Artificial Intelligence Research 70 (2021) 119-167 Submitted 03/2020; published 01/2021

Zone pAth Construction (ZAC) based Approaches for Effective Real-Time Ridesharing

Meghna Lowalekar MEGHNAL.2015@PHDCS.SMU.EDU.SG
Pradeep Varakantham PRADEEPV@SMU.EDU.SG
School of Information Systems,
Singapore Management University, Singapore

Patrick Jaillet JAILLET@MIT.EDU
Department of Electrical Engineering and Computer Science,
Massachusetts Institute of Technology, USA

Abstract

Real-time ridesharing systems such as UberPool, Lyft Line and GrabShare have become hugely popular as they reduce the costs for customers, improve per trip revenue for drivers and reduce traffic on the roads by grouping customers with similar itineraries. The key challenge in these systems is to group the "right" requests to travel together in the "right" available vehicles in real-time, so that the objective (e.g., requests served, revenue or delay) is optimized. This challenge has been addressed in existing work by: (i) generating as many relevant feasible combinations of requests (with respect to the available delay for customers) as possible in real-time; and then (ii) optimizing assignment of the feasible request combinations to vehicles. Since the number of request combinations increases exponentially with the increase in vehicle capacity and number of requests, unfortunately, such approaches have to employ ad hoc heuristics to identify a subset of request combinations for assignment.

Our key contribution is in developing approaches that employ zone (abstraction of individual locations) paths instead of request combinations. Zone paths allow for generation of significantly more "relevant" combinations (in comparison to ad hoc heuristics) in real-time than competing approaches due to two reasons: (i) Each zone path can typically represent multiple request combinations; (ii) Zone paths are generated using a combination of offline and online methods. Specifically, we contribute both myopic (ridesharing assignment focussed on current requests only) and non-myopic (ridesharing assignment considers impact on expected future requests) approaches that employ zone paths. In our experimental results, we demonstrate that our myopic approach outperforms the current best myopic approach for ridesharing on both real-world and synthetic datasets (with respect to both objective and runtime). We also show that our non-myopic approach obtains 14.7% improvement over existing myopic approach. Our non-myopic approach gets improvements of up to 12.48% over a recent non-myopic approach, NeurADP. Even when NeurADP is allowed to optimize learning over test settings, results largely remain comparable except in a couple of cases, where NeurADP performs better.

1. Introduction

Real-time taxi sharing platforms, such as UberPool, Lyft Line and GrabShare, etc. and on-demand shuttle services such as Shotl, Beeline and GrabShuttle, etc. have become hugely popular in recent years due to reduced costs for the customers and improved per trip revenue for drivers. In addition, they also help in reducing the traffic congestion as they allow customers with similar itineraries to share a vehicle. Other shared mobility services, such as car sharing, courier services, scooter sharing, bikesharing, etc. also have a similar underlying problem and the approaches (with minor extensions) presented in this paper can be used for those problems as well.

The ridesharing problem (Alonso-Mora et al., 2017a; Ma et al., 2013) is related to the vehicle routing (Ritzinger et al., 2016) and multi-vehicle pick-up and delivery problems (Parragh et al., 2008; Yang et al., 2004), where customer demand should be picked up from their origin locations and dropped at their destination locations while satisfying vehicle capacity and delay constraints. Earlier work on these problems has focussed on traditional integer programming approaches, which are limited to small scale problems of 8 vehicles and 96 requests (Ropke et al., 2007; Ropke & Cordeau, 2009). More importantly, these ridesharing problems are typically offline and do not require real-time assignment.

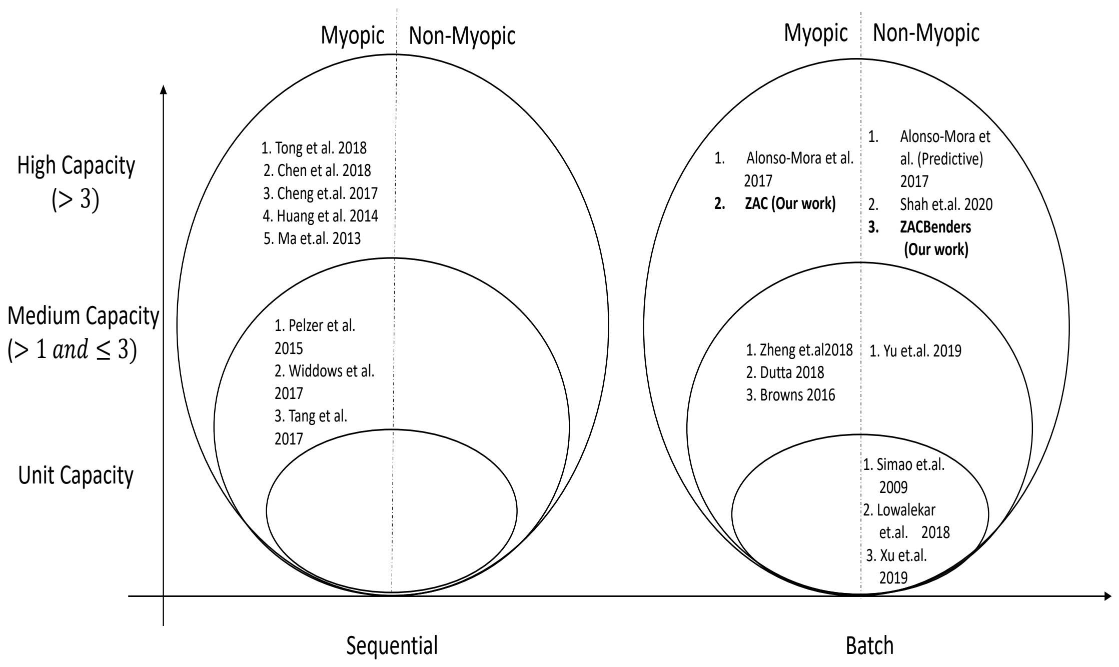


Figure 1: Related Work

In recent times, due to taxi sharing platforms, the focus has shifted to real-time taxi ridesharing problems. Many heuristic approaches have been proposed to solve these problems. As shown in the Figure 1, existing work can be categorized along three dimensions:

1. Capacity of vehicles
2. Sequential or batch consideration of requests
3. Nature of "vehicle to request combination" assignment (myopic or non-myopic).

Most existing works have considered myopic assignments (third dimension), i.e., they do not consider the future value of assignments while assigning the vehicles to current set of requests. The myopic approaches for ridesharing can further be categorized along the second dimension as:

Finding a sequential solution: Assigns one request at a time to the best available vehicle. Solution approaches are appropriate for high capacity vehicles (shuttles, buses etc.) (Tong et al., 2018; Huang et al., 2014; Ma et al., 2013) **or**

Finding a batch solution: Assigns all active requests together in a batch to the available vehicles. Most of the proposed approaches are only applicable for low capacity vehicles (e.g., taxis) (Dutta, 2018; Zheng et al., 2018).

The sequential solution is faster to compute but the quality of solution obtained is typically poor. On the other hand, the batch solution takes significantly more time to compute, but the solution quality is significantly better than the incremental solution.

As opposed to unit capacity taxi matching (first dimension), where a myopic batch solution can be computed by performing a bipartite matching between vehicles and requests, finding a myopic batch solution in multi-capacity ridesharing is challenging. This is because the underlying matching graph in multi-capacity ridesharing changes from bipartite (vehicles and requests) to tripartite (vehicles, requests, request combinations). Similar to Alonso-Mora et al. (2017a, 2017b) and Shah et al. (2020), in this paper, we focus on this most challenging category of obtaining a batch solution for high capacity vehicles in real-time.

The myopic approach by Alonso-Mora et al. (2017a) is a generalization of the greedy approach typically employed by taxi companies (Widdows et al., 2017; Tang et al., 2017; Browns, 2016) and is divided into two parts:

- The first part constructs an RTV (Request Trip Vehicle) graph. The nodes in the graph are requests, vehicles and trips. A trip in an RTV graph corresponds to a combination of requests that is feasible (with respect to the available delay for customers). There is an edge between request and trip if the request is a part of the trip. There is an edge between trip and vehicle if the vehicle can serve all the requests in that trip.

- From all allowable allocations of vehicles to trips, the second part computes an optimal allocation of vehicles to trips that minimizes delay or maximizes the number of requests served.

This approach is limited in scalability as the set of possible trips increases exponentially with the increase in the number of requests and capacity of vehicles. To ensure scalability and address the key challenge of identifying as many relevant trips as possible in real-time, this approach employs ad hoc heuristics (e.g., limiting time available and edges in RTV graph).

The non-myopic approach by Alonso-Mora et al. (2017b) is a minor extension of their myopic approach and does not provide any improvement in the number of requests served (more details in Section 6). The non-myopic approach by Shah et al. (2020) uses a similar approach as Alonso-Mora et al. (2017a) to generate the feasible trips and then uses a

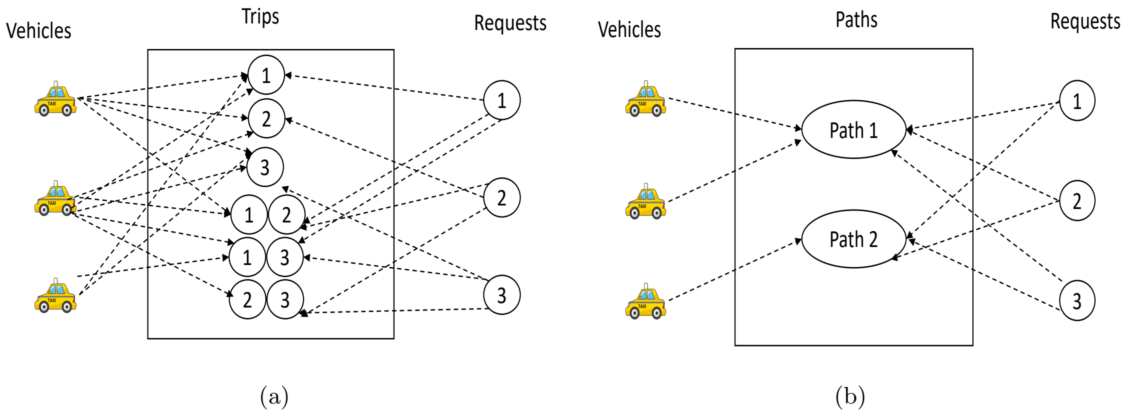


Figure 2: (a) Representation of RTV graph generated using the model by Alonso-Mora et al. (2017a) for capacity 2. (b) Representation of RPV graph generated by ZAC approach.

neural network based value function approximation to estimate the future value [1] of current assignment of vehicle to trips. While the approach provides better results than myopic approaches, it has two main issues: (1) This approach still has to employ ad hoc heuristics to identify the relevant request combinations considered for learning; and (2) Due to the need of training a separate network model for each dataset and each change of input parameter, it is not easily adaptable to different settings.

To that end, we make three major contributions in this paper. **First**, we propose a framework called ZAC (**Z**one p**A**th **C**onstruction), that is myopic and employs two crucial ideas to identify significantly more relevant trips in real-time:

- ***Focus on zone paths instead of trips:*** A zone path is a path that connects zones (a zone is an abstraction for multiple individual locations) and therefore it can group multiple trips that have "nearby" or "on the way" pick-ups and drop-offs. This focus on zone paths helps automatically capture multiple *relevant* trips with one zone path.
- ***Offline-online computation of zone paths:*** Since, we focus on zone paths, we can generate partial zone paths offline. This helps capture more relevant trips online in real-time, where the partial paths are completed.

Instead of an RTV (Request Trip Vehicle) graph in Alonso-Mora et al.'s (2017a) approach, we construct an RPV (Request Path Vehicle) graph, where we associate requests and vehicles to zone paths. This is shown in Figure 2. Once the RPV is constructed, we then employ a scalable integer linear program to find the optimal assignment (e.g., maximize revenue, maximize the number of requests served or minimize the delay) of vehicles and requests to paths.

Second, we provide a non-myopic extension of ZAC, called ZACBenders, which approximates the expected future value of assignments by considering multiple samples of future demand. By grouping the requests in demand samples based on zones and approximating

[1] As an example, for the objective of number of requests served, future value of current assignment will refer to the number of requests vehicle can serve at future timesteps as a result of being assigned to a trip at current timestep.

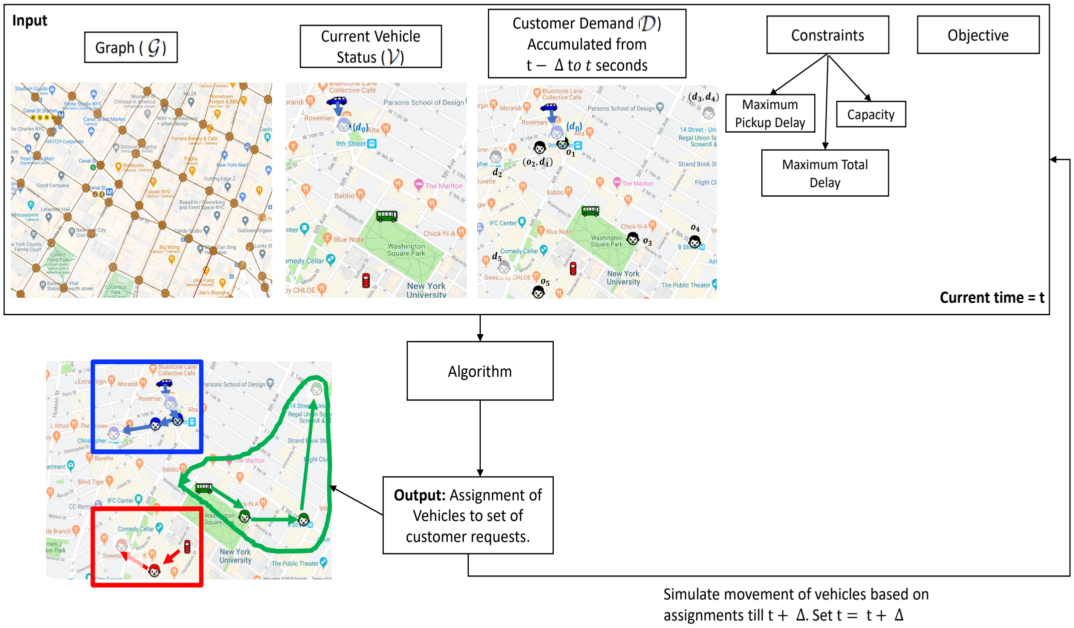

Input
Graph ($\mathcal{G}$)
Current Vehicle Status ($\mathcal{V}$)
Customer Demand ($\mathcal{D}$) Accumulated from t − Δ to t seconds
Constraints
Objective
Maximum Pickup Delay
Capacity
Maximum Total Delay
(d_0)
(d_3, d_4)
(o_2, d_1)
o_1
d_2
o_3
o_4
d_5
o_5
Current time = t
Algorithm
Output: Assignment of Vehicles to set of customer requests.
Simulate movement of vehicles based on assignments till t + Δ. Set t = t + Δ

Figure 3: RMP Description: Graph ($\mathcal{G}$) in this example is the road network of Manhattan (New York) with nodes as the road intersections. $\mathcal{V}$ provides the current status of vehicles, i.e., their location and currently assigned passengers. In the example, the blue vehicle is carrying a customer which needs to be dropped at the location represented using d_0. Customer requests are accumulated over Δ duration. The example shows 5 customer requests. The origin location of customer i is represented by o_i and destination is represented by d_i. We use a transparent person image to highlight the destination location. The algorithm should output the assignment of vehicles to customer requests while satisfying the constraints and optimizing the objective.

how future requests are served (details in the Section 4), the integer optimization in ZAC is modified to optimize the sum of values of the current assignment and expected future value over multiple demand samples. However, this results in an increase in the complexity of optimization formulation. Therefore, we propose using Benders decomposition (Benders, 1962) to break the large optimization formulation into multiple smaller problems that are solved in parallel.

Finally, we provide an exhaustive evaluation of our contributions in comparison to two leading approaches for real-time ridesharing, Trip based Formulation (TBF) (Alonso-Mora et al., 2017a) and NeurADP (Shah et al., 2020), on both real-world and synthetic datasets. We get up to 14.7% improvement over TBF and up to 12.48% over NeurADP. Even when NeurADP is allowed to optimize learning over test settings, results largely remain comparable except in a couple of cases, where NeurADP performs better. We also perform experiments on a synthetic dataset introduced by Bertsimas et al. (2018). We simulate the first and last mile transportation requests in this dataset and show that in these settings we can obtain a staggering 20% gain over TBF.

2. Rideshare Matching Problem (RMP)

Real-time ridesharing is a service provided by platforms such as Uber, Lyft etc. for arranging shared rides for multiple customers at a very short notice. Customers request a shared ride from a source to destination. The platform then groups all those requests that can share a ride – based on whether the delay[2] of reaching the destination is less than a given threshold for all requests sharing the ride. These services are popular because customers are ready to accept a delay in exchange for a reduced fare. However, after an acceptable threshold, delay can not be compensated with monetary benefits. Therefore, while grouping customer requests, platforms need to ensure that matching algorithms find groupings which do not increase the delay of individual customers beyond an acceptable threshold [3].

Figure 3 describes the RMP problem. Formally, we define RMP using the following tuple:

$$< \mathcal{G}, \Delta, \mathcal{D}, \mathcal{V}, \mathcal{C}, \tau, \lambda, \kappa, \rho, \xi^D >$$

- $\mathcal{G} = (\mathcal{L}, \mathcal{E})$ is a graph with the vertices as the set of locations. For example, as considered in previous works (Alonso-Mora et al., 2017a) the graph $\mathcal{G}$ is the road network with the set $\mathcal{L}$ including all street intersections in the road network of a city and $\mathcal{E}$ denoting the set of road segments. Figure 3 shows the visual representation of a part of the graph $\mathcal{G}$ for the Manhattan city. We assume that vehicles only pick-up and drop people off at intersections [4]. Travel Time ($\mathcal{T}$) and shortest paths ($\mathcal{S}_p$) between all location pairs in set $\mathcal{L}$ is pre-computed and stored.
- Δ denotes the decision epoch duration in seconds, i.e., the algorithm is executed every Δ seconds. In Figure 3, we show that the customer demand is collected over Δ seconds.
- $\mathcal{D}$ denotes the set of current customer requests. Each element $j \in \mathcal{D}$ is represented using the tuple: $\langle o_j, d_j, a_j \rangle$, where $o_j, d_j \in \mathcal{L}$ denote the origin and destination location and a_j denotes the arrival time of the request j. If the current decision epoch is e, then $(e-1) \cdot \Delta < a_j \leq e \cdot \Delta, \forall j$. Figure 3 shows the visual representation of customer demand by taking 5 customer requests as an example.
- $\mathcal{V}$ denotes the set of vehicles. Each element $i \in \mathcal{V}$ is represented using the tuple: $\langle \mu_i, \omega_i, q_i \rangle$. $\mu_i \in \mathcal{L}$ denotes the initial location of vehicle i, ω_i denotes the time at which vehicle first becomes available at μ_i and q_i denotes the set of customer requests assigned to vehicle i. Each element j of q_i is represented using the tuple: $\langle o_j, d_j, a_j \rangle$, where o_j, d_j, a_j are as described in the demand tuple above. Figure 3 shows the visual representation of set $\mathcal{V}$ by using three vehicles at their initial location. The blue vehicle has a previously assigned customer request.
- $\mathcal{C}$ represents the objective (e.g. revenue, number of requests served, etc.), with $\mathcal{C}^t_{ij}$ denoting the value obtained on assigning request j to vehicle i at decision epoch t.
- τ denotes maximum allowed wait time for a request (in seconds). The wait time is defined as the difference between the arrival time of a requests and the time at which vehicle picks the customer from its origin.

[2] Time taken to reach the destination using the shared ride minus the time taken using an individual ride

[3] Threshold values can be potentially learnt by surveying different customers.

[4] If the intersections are far apart and customers can be picked up/dropped off at points other than intersections, the set $\mathcal{L}$ will correspond to the set of locations where customers can be picked up/dropped off.

- λ denotes maximum allowed travel delay for requests (in seconds). The travel delay is the total delay experienced by the customer to reach its destination location. If t^d_j denotes the time at which a request j is dropped at its destination then the travel delay is given by $t^d_j - (a_j + \mathcal{T}(o_j, d_j))$.
- κ denotes the maximum capacity of each vehicle.

The goal in RMP is to assign the incoming customer requests to the vehicles such that the objective is maximized while satisfying the capacity constraints, maximum wait time and delay constraints. Figure 3 shows the assignment of customer requests to different vehicles.

The last two elements in the tuple (ρ, ξ^D) are used to represent the expected future information. The myopic approaches ignore these elements, but non-myopic approaches can use these to improve the quality of matching.

- ρ denotes the lookahead duration, i.e., the duration over which the expected future value will be computed.
- ξ^D denotes the set of samples of future customer demand with $\xi^{D,k}$ denoting the set of future requests in sample k. Each element $j' \in \xi^{D,k}$ is represented using the tuple: $\left\langle o^k_{j'}, d^k_{j'}, a^k_{j'} \right\rangle$, where $o^k_{j'}, d^k_{j'} \in \mathcal{L}$ denote the origin and destination location and $a^k_{j'}$ denotes the arrival time of the request j' in sample k. If the current decision epoch is e, then $e \cdot \Delta < a_{j'} \leq e \cdot \Delta + \rho, \forall j'$.

We first present our myopic approach ZAC in Section 3 and then in Section 4 we provide our non-myopic approach ZACBenders, which can assign incoming customer requests to the vehicles while considering future information.

3. ZAC: A Zone pAth Construction Approach for Solving RMP

Given the importance of zone path to ZAC, we first define and explain the concepts of zone and zone path. We then describe the intuitive advantages of using zone paths and then we explain the ZAC algorithm.

Definition 1 ***Zone****: refers to an abstracted location obtained by clustering locations in set $\mathcal{L}$.*

In this work, we investigated Grid Based Clustering (GBC), Hierarchical Agglomerative Clustering with Complete Linkage (HAC_MAX) and Hierarchical Agglomerative Clustering with Mean Linkage (HAC_AVG) to cluster locations into zones. We use these methods as they do not require prior knowledge about the number of clusters and have been used in earlier works on similar problems (Ma et al., 2013; Hasan et al., 2018). Please refer to Appendix A for more details on zone creation.

Definition 2 ***Zone path****: refers to an ordered sequence of nodes, where each node corresponds to either a location from set $\mathcal{L}$ or a zone.*

There are two key advantages to a zone path:

- Zone path represents multiple trips that have “nearby” or “on the way” pick-ups and drop-offs; and

- Zone path can be generated at different levels of granularity (e.g., individual locations, communities) depending on the time available.

Due to these two advantages, zone paths assist in identifying more relevant trips (combinations of requests) within a given amount of runtime. We further enhance the ability to identify more relevant trips within limited runtime, by generating zone paths partially offline and completing them in real-time depending on the set of active requests.

We generate the zone path of time span τ offline and complete the rest of the zone path online. This is because requests can be picked up only in initial τ seconds [5]. Therefore, partial zone paths generated offline automatically provide a pick-up order for the active requests. As a result, online, we only need to compute drop-off order while ensuring that the delay constraints are not violated. This contrasts with Alonso-Mora et al.'s (2017a) approach, where both pick-up and drop-off order along with the delay feasibility have to be computed online.

Intuitively, the inherent nature of zone paths to capture multiple relevant trips coupled with the extra time made available online due to offline computation of partial zone paths enables ZAC to consider significantly more relevant trips in real-time.

Due to abstraction of locations into zones, the travel time is approximately represented when considering zone paths. This can result in longer wait times or longer estimate of wait times than a path over locations in set $\mathcal{L}$. Customers prefer to have a shorter wait time pre-process (Dube-Rioux et al., 1989; Maister et al., 1984; An et al., 2019), i.e., before pick-up in this case. Therefore, it is essential to reduce this approximation in travel time computation during pick-up. We reduce this approximation by generating offline partial paths at the level of locations. This is another benefit of having an offline partial path.

ZAC is an offline-online approach for solving the RMP every few seconds on active requests and available vehicles by using offline generated partial paths. The key components of the ZAC algorithm are as follows:

- Offline: generation of all partial location paths of time span, τ from every location.
- Online: generation of RPV graph by loading and processing offline partial paths, completing the partial paths and identifying edges in RPV graph.
- Online: finding optimal assignment of requests to paths to vehicles by using an efficient integer (0/1) linear optimization

Example 1 *Figure 4 provides an example of a zone path generated using ZAC. There is a partial zone path (generated offline) over individual locations (i.e., $A \rightarrow \ldots \rightarrow F$) and the completion of that zone path (online) using larger zones (black and red).*

Example 2 *Figure 5 shows all the steps of offline-online path generation process used as part of ZAC. We will refer to these steps in Section 3.1 and 3.2.*

3.1 Offline: Partial Paths Generation

The main challenge with generating a partial path online at the level of individual locations – even for a time span of only maximum wait time, τ – is the time taken to generate all the

[5] τ is typically 300 and we experiment with values between 120-420 seconds.

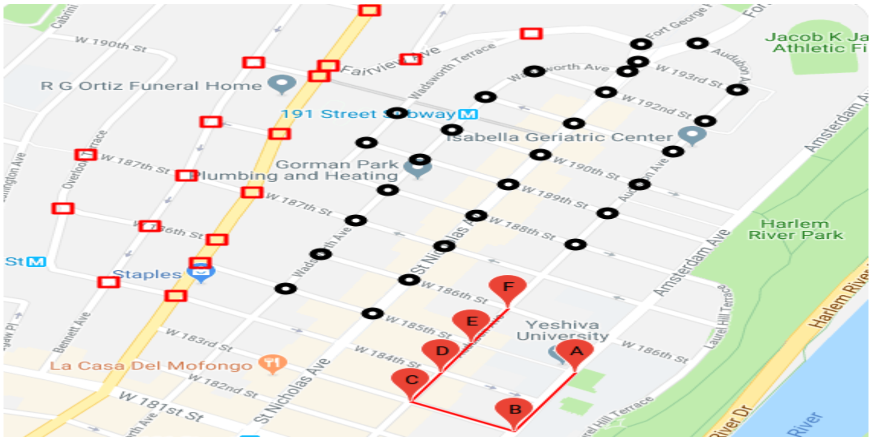


Figure 4: Example Zone Based Path. Path $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow$Black Zone $\rightarrow$ Red Zone. The part $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$ is generated offline. The decision to move from F to Black Zone then to Red Zone is taken online based on available requests.
Black Zone - Consists of Black Circled nodes.
Red Zone - Consists of Red Rectangle nodes.

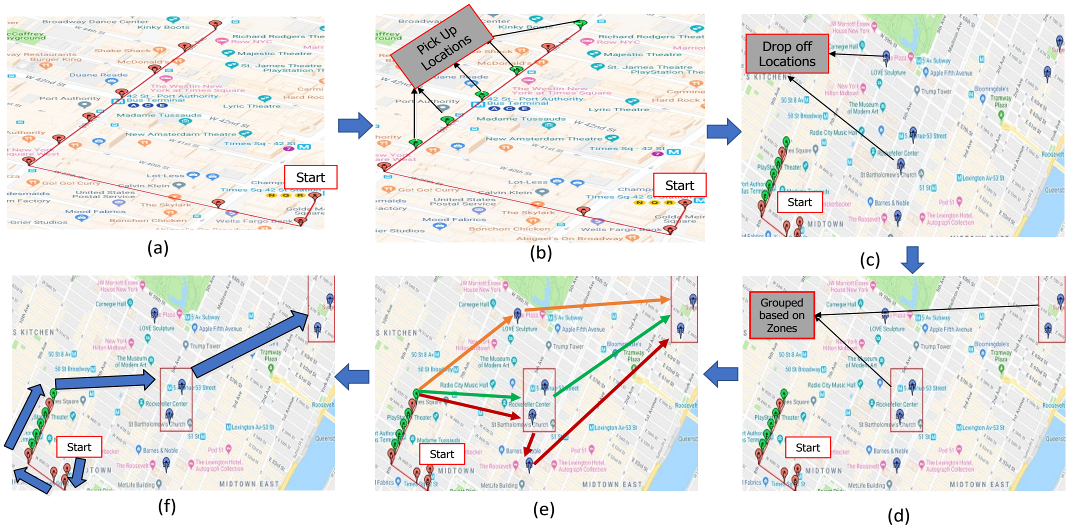


Figure 5: Example showing different steps of ZAC. (a) One of the offline partial paths. “Start” represents the start location of a vehicle. (b) Green markers represent the pick-up location of the incoming customer requests grouped along the path. (c) The blue markers represent the destination locations of the requests which are grouped along the path based on their pick-up location. (d) The destination locations are grouped together based on different zones (e) The path is completed by starting exhaustive search at the last green marker. The exhaustive search generates three different paths shown in orange, green and red colour. (f) The path represented by blue big arrows shows one of the complete zone paths starting at the initial location of the vehicle.

paths. Therefore, we compute these partial paths (of span τ seconds) offline by generating all simple paths of duration τ in the network $\mathcal{G}$. The number of all possible paths grows exponentially with the increase in the value of τ and increase in the number of locations. In case all possible paths can not be generated due to memory constraints, we can employ a data-driven approach (based on historical data) to generate paths that have high likelihood of grouping a large number of requests. More details about the data-driven approach is present in Appendix B.

Figure 5(a) shows one of the offline generated partial paths as an example starting at the location denoted by "Start" and visiting all the locations represented by red markers.

The start point of each offline partial path is one of the locations from the set $\mathcal{L}$ and the end point can be any of the locations which is reachable within τ duration from the start location. These offline partial paths ($\mathcal{P}_{off}$) are stored by indexing on the start location and start time ($\mathcal{P}_{off}[l,t]$) [6]. The start time associated with the path indicates the time at which the first node (location) in the path is visited. For a clear explanation, two paths starting at the same location but having different start times are considered different. This is because vehicles can become available at the same location but at different time. These offline partial paths are further indexed by the location and time of each node present in the path for quick online processing.

Algorithm 1 ZAC-Online()

```
$t = starttime$ (in seconds)
$\mathcal{P}_{off} = \bigcup_{l \in \mathcal{L}, t' < \tau} \mathcal{P}_{off}[l, t']$ = LoadOfflinePartialPaths()
$\mathcal{T}$ =LoadTravelTimes(), $\mathcal{S}_p$ =LoadShortestPaths()
while $t < endtime$ do
    $t_1 = t - starttime$
    if $(t_1)\%\Delta == 0$ then
        $\mathcal{D}, \mathcal{V} \leftarrow$GetCurrentDemand-VehicleStatus($t$)
        $\mathcal{P}, Pv, Pr, b, N$ = GenerateRPVGraph($t, \mathcal{P}_{off}, \mathcal{D}, \mathcal{V}, \mathcal{T}, \mathcal{S}_p$)
        SolveOptimization($\mathcal{P}, Pv, Pr, b, N$)
    UpdateVehicleStatus()
    $t = t + 1$
```

3.2 Online

We now describe the crucial online component of ZAC that generates the RPV graph and finds the optimal match on the generated RPV graph. The pseudocode for the online component ZAC-Online is provided in Algorithm 1. After loading the offline computed partial paths, travel times and shortest paths, at every decision epoch, ZAC-Online considers the currently available batch of requests and current vehicle status to find the optimal assignment in two steps: (1) Generation of the Request, Path and Vehicle (RPV) graph and (2) Finding optimal match in RPV graph using a linear integer optimization model. We now describe the two steps of ZAC in detail.

[6] We discretize the time at the level of 10 seconds.

3.2.1 Generation of the RPV Graph

As shown in Algorithm 2, there are three key steps to RPV graph generation: (1) Online processing of Offline Partial Paths; (2) Online Partial Zone Path Completion; (3) Identifying edges in the RPV graph.

Algorithm 2 GenerateRPVGraph(t, $\mathcal{P}_{off}$, $\mathcal{D}$, $\mathcal{V}$, $\mathcal{T}$, $\mathcal{S}_p$)

```
$\mathcal{P}'_{off}, \mathcal{R}'$ = ProcessOfflinePartialPaths($t$, $\mathcal{P}_{off}$, $\mathcal{D}$, $\mathcal{V}$,$\mathcal{T}$, $\mathcal{S}_p$)
$\mathcal{P}, \mathcal{R}''$ = OnlineCompletion($t, \mathcal{P}'_{off}, \mathcal{R}', \mathcal{T}, \mathcal{S}_p$)
$\mathcal{P}, Pv, Pr, b, N$ = IdentifyEdgesRPVGraph($t$, $\mathcal{P}$, $\mathcal{D}$, $\mathcal{V}$, $\mathcal{R}''$)
return $\mathcal{P}, Pv, Pr, b, N$
```

Algorithm 3 ProcessOfflinePartialPaths(t,$\mathcal{P}_{off}$,$\mathcal{D}$,$\mathcal{V}$, $\mathcal{T}$,$\mathcal{S}_p$)

```
$\mathcal{P}'_{off} = []$,$Lt_v = \bigcup_{i \in \mathcal{V}} (\mu_i, \omega_i)$
Create $H$ threads. Each thread $h$ processes $\mathcal{P}^h_{off} = \bigcup_{k' \in Lt^h_v} \mathcal{P}_{off}[k']$, ,s.t., $Lt^a_v \cap Lt^b_v = \phi, \forall a \neq b$ and $\cup_h Lt^h_v = Lt_v$
for each thread $h$ do
    $\mathcal{V}' \subset \mathcal{V}, s.t., \forall i' \in \mathcal{V}', (\mu_{i'}, \omega_{i'}) \in Lt^h_v$
    for $j \in \mathcal{D}$ do
        $\mathcal{P}^{h,j}_{off}$ = GetPathsFromIndex($\mathcal{P}^h_{off}, o_j, a_j - t, a_j - t + \tau$)
        for each path $k \in \mathcal{P}^{h,j}_{off}$ do
            $lb_j = a_j - t + \mathcal{T}(o_j, d_j), ub_j = lb_j + \lambda$
            if $\mathcal{R}[k]$ contains $d_j$ then
                $\mathcal{R}[k][d_j][1] = max(\mathcal{R}[k][d_j][1], ub_j)$
            else
                $\mathcal{R}[k].add(d_j, (lb_j, ub_j))$
            $\mathcal{R}_p[k].add(o_j)$
            if $lb_j < \tau$ and $k$ visits $d_j$ then
                $\mathcal{R}_p[k].add(d_j)$
            else if $ub_j < \tau$ and $k$ does not visit $d_j$ then
                $\mathcal{R}[k].remove(d_j, (lb_j, ub_j))$
    for $i \in \mathcal{V}'$ do
        $\mathcal{R}[k], \mathcal{R}_p[k]$ = GetPathsForVehicle($i, q_i, \mathcal{R}[k], \mathcal{R}_p[k], \mathcal{P}_{off}$)
    for each path $k$ do
        if $|\mathcal{R}_p[k]| > 0$ then
            Remove nodes not in $\mathcal{R}_p[k]$, update $\mathcal{R}_k$ using $\mathcal{T}$, $\mathcal{S}_p$
            $\mathcal{P}'^h_{off}.add(k)$
for each thread $h$ do
    $\mathcal{P}'_{off}.addAll(\mathcal{P}'^h_{off})$
return $\mathcal{P}'_{off}, \mathcal{R}$
```

Online Processing of Offline Partial Paths: The offline generated partial paths are processed online based on the pick-up locations of the currently available requests and current status of all vehicles (location of vehicles, currently assigned requests to vehicles).

Figure 5(b) and Figure 5(c) shows the online processing of the offline partial path based on the currently available requests. In Figure 5(b), the green markers represent the pick-up location of the requests which can be grouped using the offline partial path and in the Figure 5(c), we use the blue markers to represent the destination locations of the requests which had pick-up at one of the locations represented using green marker. We also store a lower and upper bound on the time by which each of the blue marker should be visited for the delay constraints to be satisfied. Therefore, by online processing of offline partial paths, for each path, we get the set of requests which can be grouped based on their pick-up location and we also get the order in which the locations should be visited. The detailed algorithm is shown in Algorithm 3.

The algorithm takes as an input the set of all offline generated partial paths ($\mathcal{P}_{off}$), incoming customer requests ($\mathcal{D}$), and current status of all vehicles ($\mathcal{V}$). The output of the algorithm is the set of offline partial paths ($\mathcal{P}'_{off} \subset \mathcal{P}_{off}$) which can serve at least one request from the set $\mathcal{D}$. For each of these paths, the algorithm also provides the information about the requests grouped along the path ($\mathcal{R}'$). The information includes the destination location and the lower and upper bound on the time by which the location needs to be visited. The individual steps of the algorithm are described below.

Steps 1-2 of the algorithm ensures that we consider only those paths that start at a location and time where at least one vehicle is present, and these paths are processed in parallel using multiple threads. The GetPathsFromIndex function returns the set of offline partial paths that visit the given location within a given time interval and uses the precomputed offline indexes for quick online retrieval. Step 12 stores the set of destination locations of the currently available requests grouped along the path (based on the pick-up). In addition to the destination location, we also store the lower and upper bound on the time by which the location should be visited. Similarly, in step 19, we store the destination locations of the requests previously assigned to vehicles. In step 19, we consider only those paths that can potentially satisfy all the previously assigned requests for a vehicle. This is because a vehicle will be assigned to a path if and only if it can serve all the previously assigned requests. In addition, a vehicle should deviate from its current path only if it can be assigned to a new request, therefore, we consider only those paths which can pick at least one of the newly available requests.

Steps 14 and 19 ensure that if the drop-off location of request can be visited in the partial path, then it is considered in the processing.

In the end, in steps 20-22, as an optimization, we only keep those locations in the partial paths that correspond to a pick-up or drop-off location and update the travel time and path between the locations using $\mathcal{T}$ and $\mathcal{S}_p$.

The offline generated partial paths significantly improve the scalability of completing the path online using exhaustive search. This provides more time online for considering more zone paths and hence more relevant trips.

Online Partial Zone Path Completion: The subset of offline partial paths obtained from the algorithm used in the previous step (Algorithm 3) are completed online in this step using exhaustive search starting at the last location in the partial path. For the example in Figure 5, we complete the offline partial path by starting the exhaustive search at the last green marker. The search space is the set of destination locations represented by blue markers. To reduce the computational complexity, we group nearby destination locations using zones. Figure 5(d) shows that the destination locations of the requests which are nearby are grouped using zones. Figure 5(e) shows that we get multiple zone paths by completing a single offline partial path using exhaustive search. We use orange, green and red colour arrows to highlight three different paths. Finally, we highlight one of the completed zone paths (out of the three possible paths) starting at the location "Start" in Figure 5(f) using big blue arrows.

The complete process is formally shown in the Algorithm 4. The output of Algorithm 3 serves as an input to the Algorithm 4, i.e., the algorithm takes as an input, the set of offline partial paths ($\mathcal{P}'_{off}$) and the information about requests associated with each of the offline partial path ($\mathcal{R}'$). The output of the algorithm is the set of completed zone paths ($\mathcal{P}$). For each of these zone paths in set $\mathcal{P}$, the algorithm also outputs the set of requests which can potentially be served using the zone path ($\mathcal{R}$). The detailed steps of the algorithm are explained below.

As the offline partial paths are independent of each other, to speed up the path generation process, we perform the online path completion of the offline partial paths $\mathcal{P}'_{off}$ in parallel by creating multiple threads as shown in the pseudocode provided in Algorithm 4. To complete each offline partial path, we need to consider the destination locations of all the requests associated with the path. As we also have a lower and upper bound on the time by which the destination locations of the requests should be visited, we can prune the search space by exploring only those branches in the search tree where these time limits are satisfied.

The computational complexity of online partial path completion is dependent on the number of destination locations (size of $\mathcal{R}'[k]$ in Algorithm 4) and can be significant, therefore, we use zones (and not individual locations) in this step. As mentioned before, by using zones, travel time is approximately represented, which can result in additional delay for requests. The additional delay introduced is dependent on the size of the zones chosen. The size of the zone is defined as the time taken to travel within a zone. Zone size 0 indicates that locations in set $\mathcal{L}$ are used.

Therefore, to consider a trade-off between computational complexity and the quality of a solution, we propose picking the zone sizes dynamically for each offline partial path. In order to fix the amount of dynamism in zone size, we use a parameter M that defines the number of different zone sizes that can be used in completion of offline partial paths. $M = 1$, implies static zone sizes, i.e., using zones of a fixed size for online completion of all offline partial paths. The zones of M different sizes are generated offline and in the step 6, depending on the number of destination locations and M available zone sizes, we decide the appropriate zone size for the partial path k [7]. Please note that the exhaustive search in step 8, will return multiple completed zone paths corresponding to a single partial path k as shown in the example (Figure 5) before.

[7] In the experiments, we use $M = 4$ with zone sizes 0,60,120,300 and use the zone size that reduces the number of locations to 12 (this provides the best trade-off between runtime and solution quality and is determined based on experiments.)

Algorithm 4 OnlineCompletion($t, \mathcal{P}'_{off}, \mathcal{R}', \mathcal{T}, \mathcal{S}_p$)

```
 1: $\mathcal{P} = [], \mathcal{R} = []$
 2: Create $H$ threads.
    Each thread $h$ processes $\mathcal{P}'^h_{off} \subset \mathcal{P}'_{off}$ ,s.t., $\mathcal{P}'^h_{off} \cap \mathcal{P}'^{h'}_{off} = \phi, \forall h \neq h'$ and $\cup_h \mathcal{P}'^h_{off} = \mathcal{P}'_{off}$
 3: for each thread $h$ do
 4:     $\mathcal{P}^h_{on} = [], \mathcal{R}^h_{on} = []$
 5:     for each path $k$ do
 6:         $z = $ getAppropriateZoneSize$(\mathcal{R}'[k], M)$
 7:         $\mathcal{R}' = $ convert$(\mathcal{R}'[k], z)$
 8:         $\mathcal{P}^h_{on}, \mathcal{R}^h_{on} = $ ExhaustiveSearch$(end\_node(k), \mathcal{R}', \mathcal{P}^h_{on}, \mathcal{R}^h_{on})$
 9: for each thread $h$ do
10:     $\mathcal{P}.addAll(\mathcal{P}^h_{on})$
11:     $\mathcal{R}.addAll(\mathcal{R}^h_{on})$
12: return $\mathcal{P}, \mathcal{R}$
```

For the objective of maximizing the number of requests served, the paths that start at the same location at the same time and serve a subset of requests served by another path are redundant. This is because we check for capacity constraints in the optimization formulation presented in the next section. So, a single path serving r requests can be used to represent all request combinations, $\sum_{i=1}^{r} \binom{r}{i}$. Therefore, the search tree in step 8 of Algorithm 4 can be pruned appropriately to search only for non-redundant paths. This reduces the size of set $\mathcal{P}$, which, in turn reduces the complexity of the optimization formulation presented in the Section 3.2.2.

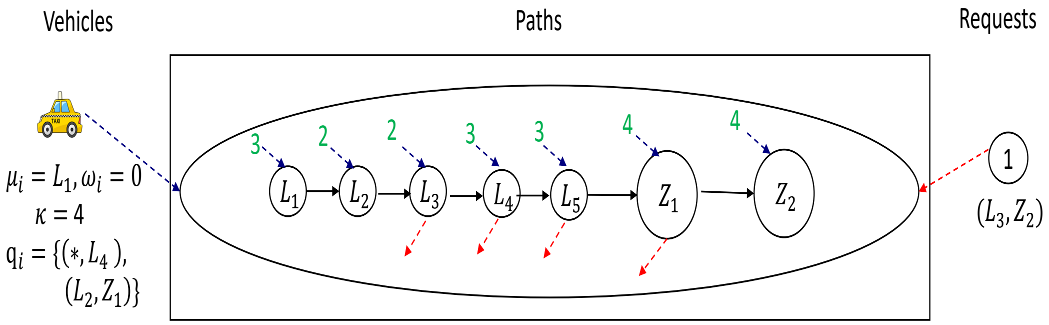


Figure 6: Representation of assignment of vehicle and request to a zone path

Identifying Edges in the RPV Graph: Once the zone paths ($\mathcal{P}$) are created using the offline-online method described above, we construct the RPV (Request Path Vehicle) graph by finding the set of requests and vehicles that can be assigned to each of the generated zone paths. We use the information available from the previous 2 steps about the requests and vehicles that can be assigned to these zone paths and process the paths in parallel using multiple threads to speed up the computation. This step is essential as in Algorithm 3, when the same destination location has a different value for the upper limit on time in step 10, we take the maximum value. Therefore, in the path generated using Algorithm 4, the delay constraint may be violated for some requests in such cases.

In this step, we ensure that a request is assigned to a zone path, if and only if, the path visits the pick-up and drop-off location of a request within the delay constraints. The binary constants b (defined in Table 1 and used in optimization formulation presented next) are also populated in this step. A vehicle i represented by the tuple (μ_i, ω_i, q_i) can be assigned to a zone path if the initial location of the vehicle, μ_i, is same as the starting location of the path, the start time of the path is same as the availability time of vehicle ω_i and the currently assigned set of requests, q_i, can be served using the path. The vehicle capacity κ along with q_i is used to compute the number of free seats (N) in the vehicle at each zone/location.

Example 3 *Figure 6 shows a graphical view of the same for a single vehicle, request and path. The path is represented using a sequence of locations/zones in the order in which they will be visited. In Figure 6, we use blue arrows to denote the incoming flow by vehicle i assignment and green numbers indicate the number of free seats at the location/zone for vehicle i. The number of free seats is computed by taking κ and q_i of the vehicle into consideration. In figure, in the representation of q_i, we use * to indicate that customer is already present in the vehicle and provide its drop-off location. The red arrows indicate outgoing flow by request assignment.*

At each location, the optimization formulation presented next will ensure that the outgoing flow (the number of requests assigned) is less than or equal to the incoming flow (total number of free seats in the vehicles assigned to the path), i.e., at each location/zone, the capacity constraints are satisfied.

3.2.2 Finding Optimal Match in RPV Graph

We now describe the integer linear programming optimization formulation to optimize the assignment of requests and vehicles to zone paths. $\mathcal{P}$ denotes the set of zone paths generated in the previous step. $\mathcal{P}_m^n$ is used to denote the n^{th} location/zone in zone path m. Let $Pr_j \subset \mathcal{P}$ denotes the set of paths that can serve request j while satisfying delay constraints. Similarly Pv_i denotes the set of paths that can be assigned to vehicle i based on its current location μ_i, availability time ω_i and already assigned/picked-up requests q_i. Binary constants b_{jm}^n are set to 1 if the pick-up location of request j is visited but drop-off location/zone is not visited along path m by the n^{th} location/zone. These are computed as part of generation of RPV graph as shown in the previous section. Table 1 describes the notation used in the optimization formulation.

The objective of the optimization formulation described in Table 2 is to maximize the number of served requests. Constraints (2) and (3) ensure that each vehicle and each request is assigned to at most one path. Constraint (4) ensures that for every path at every location/zone, the capacity constraints are satisfied. The capacity constraints can be violated only while picking up a new request, therefore, the constraint (4) is redundant for the locations/zones visited after τ duration.

The formulation is run at every decision epoch, i.e., after every Δ seconds. The solution of the optimization formulation provides assignment of vehicles and requests to paths.

Variable	Description
x_{jm}	Binary variable denoting if the request $j \in \mathcal{D}$ is assigned to path m.
y_{im}	Binary variable denoting if the vehicle i is assigned to the path m.
Pv_i	$Pv_i \subset \mathcal{P}$ denotes the set of paths that can be assigned to vehicle i based on its current status μ_i, ω_i and q_i.
Pr_j	$Pr_j \subset \mathcal{P}$ denotes the set of paths that can be assigned to request $j \in \mathcal{D}$.
b^n_{jm}	Binary constant: 1 if $\exists n' : n > n'\ P^{n'}_m = o_j$ && $\nexists n'' : n'' < n, n'' > n'\ P^{n''}_m == d_j$
$N(i,m,n)$	Number of free seats in the vehicle i for path m at n^{th} location/zone.

Table 1: Notations

Using these assignments, we can perform the assignment of requests to vehicles. The paths assigned to vehicles are also updated to keep only those locations that correspond to pick-up or drop-off location of assigned requests and update the travel time and path between the locations using $\mathcal{T}$ and $\mathcal{S}_p$.

Once a vehicle is assigned to a set of requests at any decision epoch, the assignment is not changed but the path of vehicle can change at next decision epoch to accommodate additional requests. The current set of requests assigned to a vehicle, q_i, limits the number of paths to which it can be assigned in subsequent decision epochs. The number of free seats in vehicle i for path m at location/zone n, $N(i,m,n)$ is computed based on κ and q_i (as shown in Figure 6) and is 0 if $m \notin Pv_i$.

Similar to Alonso-Mora et al. (2017a), we perform a rebalancing of unassigned vehicles to high demand areas at the end of optimization formulation.

4. ZACBenders: A Non-myopic Approach for Solving RMP

In this section, we first present the challenges in solving RMP with future information (represented using samples, ξ^D in the RMP model) and then present our non-myopic approach ZACBenders. The potential samples, ξ^D can be obtained by considering the demand observed in the past data. After considering future information, the goal is to find the assignment of vehicles to requests that maximizes the sum of objective value at the current decision epoch and the expected objective value for the future decision epochs.

4.1 Challenges in Solving RMP with Future Information

As shown in the Figure 7, to solve RMP with future information (ξ^D) we need to assign vehicles and request to the zone paths at each decision epoch and for each sample ($\xi^{D,k}$). The state (i.e., location and requests being served) of vehicle at each decision epoch should be updated based on the assignments (obtained by solving the RPV graph) at the previous decision epochs. The paths at future decision epochs for each sample need to be generated by considering the requests present in the sample and the optimization problem should be updated to consider the assignments at future decision epochs for all the samples.

There are two major bottlenecks in the above process.

SolveOptimization($\mathcal{P}, Pv, Pr, b, N$):

$$\max \quad \sum_{j \in \mathcal{D}} \sum_{m \in Pr_j} x_{jm} \tag{1}$$

$$s.t. \quad \sum_{m \in Pr_j} x_{jm} \leq 1 ::: \forall j \in \mathcal{D} \tag{2}$$

$$\sum_{m \in Pv_i} y_{im} \leq 1 ::: \forall i \in \mathcal{V} \tag{3}$$

$$\sum_{j \in \mathcal{D}} x_{jm} \cdot b^n_{jm} \leq \sum_i y_{im} \cdot N(i,m,n) ::: \forall m \forall n \tag{4}$$

$$x_{jm}, y_{im} \in \{0,1\} \forall i,j,m \tag{5}$$

Table 2: Optimization Formulation for ZAC

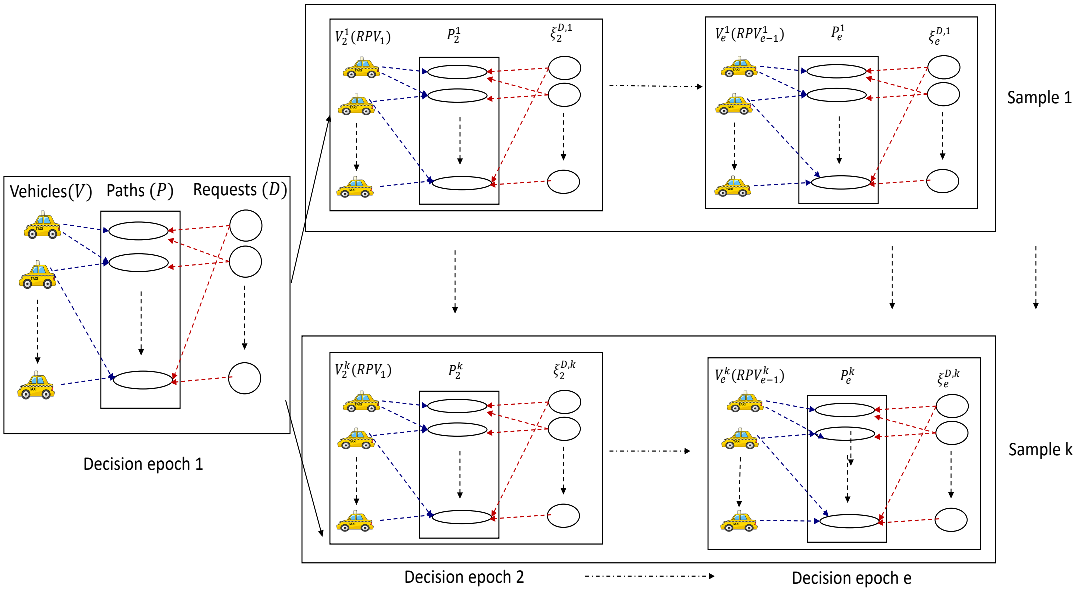


Figure 7: Assignment of vehicles and requests to zone paths over multiple samples of future demand. We use RPV_e^k to denote the RPV graph for decision epoch e in sample k. For the special case of first decision epoch, we denote the graph by RPV_1. $\mathcal{P}_e^k$ denote the set of paths generated for decision epoch e in sample k. $\xi_e^{D,k}$ denote the set of requests available at decision epoch e in sample k. $\mathcal{V}_e^k(RPV_{e-1}^k)$ denote the state of vehicles at decision epoch e in sample k as a result of assignments obtained by solving the RPV graph at previous decision epoch in the same sample. Therefore, at each decision epoch for each sample, it is a matching in a tripartite graph (with vehicles, paths and requests as three parties).

1. As described in the Section 3, the path generation process for a single decision epoch is challenging, so generating paths in real-time after considering requests in each

sample for all future decision epochs for all possible updates to the vehicles states is computationally intractable.

2. The optimization formulation of assigning vehicles and requests to zone paths is an integer optimization problem. Including requests for all the samples at future decision epochs in this integer optimization formulation increases the number of variables and constraints, which makes it difficult to solve online in real-time.

4.2 ZACBenders Approach

The overall flow of ZACBenders is similar to ZAC with the only difference in the step of finding the optimal assignment of vehicles and requests to zone paths. Table 3 highlights the difference between ZAC and ZACBenders approach. ZACBenders considers future information in the step of finding the optimal assignment of vehicles and requests to zone paths. As mentioned in Section 4.1, incorporating future information makes the problem challenging, therefore, we first provide a two-stage stochastic approximation, ZACFuture to handle these challenges. To efficiently solve the ZACFuture optimization formulation in real-time, we employ Benders Decomposition.

ZAC	**ZACBenders**
Offline	
1. Generation of all partial location paths of time span, τ from every location.	1. Same
Online	
1. Generation of RPV graph 2. Finding optimal assignment of requests and vehicles to zone paths by using the (0/1) integer optimization in Table 2.	1. Same 2. Process the requests in $|\xi^D|$ samples to generate the second stage of the proposed two-stage approximation. 3. Follow the steps in Figure 9 to find the optimal assignment of requests and vehicles to zone paths while considering future information.

Table 3: Differences between ZAC and ZACBenders

4.2.1 Two-Stage Stochastic Approximation

As mentioned in Section 4.1, it is difficult to solve RMP with future information by generating paths considering requests in all samples, therefore, we propose a two-stage stochastic approximation [8]. The first stage assigns vehicles and requests available at the current decision epoch to the zone paths. In the second stage, for each sample, instead of solving a matching problem in a tripartite graph (with vehicles, paths and requests as three parties)

[8] We have experimented with many other approximations such as considering simple extensions of the existing zone paths to include request in samples and then assigning requests in samples to these extended paths but we describe in detail the approximation that worked best in practice. We acknowledge that it is possible to improve the performance even more by designing better approximations.

at each future decision epoch, we solve a weighted bipartite matching between vehicles and requests available for assignment at all future decision epochs. The matching problem in tripartite graph is NP-hard, but the weighted bipartite matching is polynomial-time solvable, therefore, this approximation makes the second stage problem simpler. Specifically, we employ a decomposition of the resultant optimization problem (more details in Section 4.2.3) to get real-time performance.

We now describe the approximations which allow us to simplify the second stage problem for each sample by modelling it as a weighted bipartite matching problem.

- *Approximation 1:* Instead of using exact locations from set $\mathcal{L}$, we use abstracted locations, i.e., zones. The origin/destination of each request in the sample and the location of each vehicle is mapped to the zones.
- *Approximation 2:* A vehicle will serve requests in samples (ξ^D) only after it finishes serving all the currently assigned requests. That is to say, we ignore that any future request can be inserted in the vehicle's path and as a result a vehicle is considered available again for assignment only after it reaches the end of zone path generated in the first step.
- *Approximation 3:* Requests in samples (ξ^D) can be assigned to the same vehicle if and only if they have identical origin zone, identical destination zone and the decision epoch at which they become available for the assignment is also the same.

These approximations help in reducing the complexity of the problem, but they still allow us to get a good estimate of the future because of the following reasons:

- The second approximation ensures that the vehicles that are considered for assignment at second stage are empty, i.e., they do not have any request assigned to them. So when the assignment optimization problem is solved (with limited lookahead duration of ρ), instead of assigning the vehicle to a longer duration path (which keeps it occupied for more than ρ duration), it will assign the vehicle to those shorter duration zone paths (in the first stage) that redirect it to zones where future requests are present. At any decision epoch, it is easier to assign multiple requests to an empty vehicle as compared to a vehicle that has a passenger on board. Therefore, despite ignoring that future requests can be picked up before dropping all currently assigned requests, this provides a good approximation.
- The third approximation (along with first approximation) ensures that the requests are grouped when they have nearby pick-up and drop-off locations. Though we will miss grouping the requests that have on the way pick-ups/drop-offs, by making this approximation and using an appropriate zone size, we will still be able to implicitly consider a subset of possible paths.

Formally, we map the origin and destination location of requests in the future decision epochs to zones of size [9] Z^s and the elements in $\xi^{D,k}$ are grouped together based on the origin/destination zone and decision epoch. After grouping, each element j' of $\xi^{D,k}$ is represented using tuple $\left\langle o^{z,k}_{j'}, d^{z,k}_{j'}, e^k_{j'}, \eta^k_{j'} \right\rangle$ where $o^{z,k}_{j'}$ denotes the origin zone of the element j' in sample k, $d^{z,k}_{j'}$ denotes the destination zone of the element j' in sample k, $e^k_{j'}$ denotes the decision epoch at which element j' of sample k will be considered for assignment and

[9] As mentioned before, the size of the zone is defined as the time taken to travel within a zone.

$\eta^k_{j'}$ denotes the number of requests with origin $o^{z,k}_{j'}$, destination $d^{z,k}_{j'}$ at decision epoch $e^k_{j'}$ in sample k. We use $\mathcal{A}$ to denote the set containing all possible pairs of zones of size $\mathcal{Z}^s$ and decision epochs e' such that if e is the current decision epoch then $e < e' \leq \lfloor \frac{\rho}{\Delta} \rfloor$. Each vehicle is mapped to an element in set $\mathcal{A}$ and is assigned requests in samples by using above approximations.

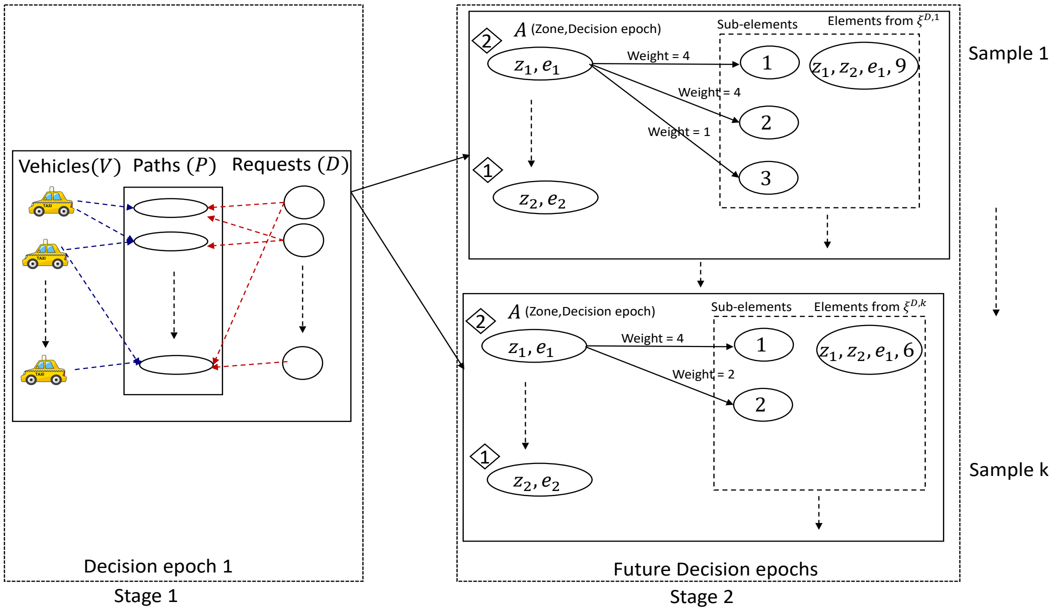


Figure 8: Two-Stage stochastic approximation for assignment of vehicles and requests to zone paths over multiple samples of future demand (For $\kappa = 4$). The zone and decision epoch mentioned in the oval are the zone and decision epoch at which the paths at the first stage in set $\mathcal{P}$ ends. Therefore, vehicles have dropped all the assigned requests from set $\mathcal{D}$ (in first stage), once they reach the zone and decision epoch present in the oval. The number inside a diamond represents the number of empty vehicles present in the zone and decision epoch mentioned in the oval box. At the second stage, for each sample, a bipartite matching is performed between empty vehicles and available requests for all future decision epochs as compared to the matching in a tripartite graph (with vehicles paths and requests as three parties) for each sample in each decision epoch as shown in Figure 7.

The assignment of vehicles to paths at first stage determines the zone and the decision epoch at which the vehicles will become available again for assignment (Approximation 2), and as all the vehicles have identical maximum capacity [10], in the second stage, we can group the vehicles based on the zone and the decision epoch at which they become available again for assignment. A vehicle can be assigned to a request if and only if it can reach the origin location of request within the maximum allowed wait time, i.e. τ. Let $\mathcal{E}^k$ denotes the set of all such assignment edges between vehicles and requests for sample k. To ensure that a vehicle can be assigned at most κ requests and these requests can be grouped together as

[10] In the experiments, we show that even if all vehicles do not have identical maximum capacity, the approximation still works well. In that case, in the second stage, we take the maximum capacity of each vehicle as an average of all vehicle's maximum capacity.

per Approximation 3, if $\eta^k_{j'} > \kappa$, we divide the element into $\lceil \frac{\eta^k_{j'}}{\kappa} \rceil$ subelements of element j' and allow each subelement to be assigned at most once, but if a vehicle of type i' is assigned to r^{th} subelement of element j' in sample k, the weight received is given by

$$w^k_{i'j'r} = \begin{cases} 0 \quad if(i',j',r) \notin \mathcal{E}^k \\ \kappa \quad if(i',j',r) \in \mathcal{E}^k \ and \ r < \lfloor \frac{\eta^k_{j'}}{\kappa} \rfloor \\ \eta^k_{j'} \,\%\, \kappa \quad otherwise \end{cases}$$

Therefore, this creates a bipartite graph with one side containing vehicles grouped based on their type (i.e., zone and decision epoch at which they become available based on the path assigned in the first stage) and the other side containing the request groups (all subelements of the element $j' \in \xi^{D,k}, \forall j'$).

Figure 8 shows the graphical representation of the two-stage stochastic approximation, where first stage performs the matching in a tripartite graph (with vehicles, requests and zone paths as three parties), and in the second stage within each sample, there is a bipartite matching. We now describe the optimization formulation that can be used to solve this two-stage stochastic approximation.

ZACFuture($\mathcal{P}, Pv, Pr, b, N, \xi^D$):

$$\max \quad \sum_{j\in\mathcal{D}} \sum_{m\in Pr_j} x_{jm} + \frac{1}{|\xi^D|} \sum_{k=0}^{|\xi^D|} \sum_{j'\in\xi^{D,k}} \sum_{r=0}^{\lceil \frac{\eta^k_{j'}}{\kappa} \rceil} \sum_{i'\in\mathcal{A}} w^k_{i'j'r} \cdot u^k_{i'j'r} \tag{6}$$

$$s.t. \quad \sum_{m\in Pr_j} x_{jm} \leq 1 ::: \forall j \in \mathcal{D} \tag{7}$$

$$\sum_{i'\in\mathcal{A}} u^k_{i'j'r} \leq 1 ::: \forall j' \in \xi^{D,k}, 0 \leq r < \lceil \frac{\eta^k_{j'}}{\kappa} \rceil, \forall 0 \leq k < |\xi^D| \tag{8}$$

$$\sum_{m\in Pv_i} y_{im} \leq 1 ::: \forall i \in \mathcal{V} \tag{9}$$

$$\sum_{j\in\mathcal{D}} x_{jm} \cdot b^n_{jm} \leq \sum_i y_{im} \cdot N(i,m,n) ::: \forall m \forall n \tag{10}$$

$$\sum_{j'\in\xi^{D,k}} \sum_{r=0}^{\lceil \frac{\eta^k_{j'}}{\kappa} \rceil} u^k_{i'j'r} \leq \sum_{i\in\mathcal{V}} \sum_{m;f(m)=i'} y_{im} ::: \forall i' \in \mathcal{A} \tag{11}$$

$$y_{im} \in \{0,1\} ::: \forall i \in \mathcal{V}, m \in \mathcal{P} \tag{12}$$

$$x_{jm} \in \{0,1\} ::: \forall j \in \mathcal{D}, m \in \mathcal{P} \tag{13}$$

$$u^k_{i'j'r} \in \{0,1\} ::: \forall i' \in \mathcal{A}, j' \in \xi^{D,k}, 0 \leq r < \lceil \frac{\eta^k_{j'}}{\kappa} \rceil, 0 \leq k < |\xi^D| \tag{14}$$

Table 4: Optimization Formulation for Two-Stage stochastic approximation of RMP with future samples

4.2.2 ZACFuture: Optimization Formulation to Solve the Two-Stage Stochastic Approximation

In this section, we describe the optimization formulation ZACFuture to solve the two-stage stochastic approximation presented in the previous section. Table 4 presents the optimization formulation ZACFuture, which maximizes the number of requests served for the current decision epoch and the expected number of requests served over future demand samples.

We use $u^k_{i'j'r}$ to denote the assignment of vehicle of type i' (i.e., the vehicles present in the zone $z_{i'}$ at decision epoch $e_{i'}$, where $(z_{i'}, e_{i'})$ is the tuple representation of element $i' \in \mathcal{A}$) to the r^{th} subelement of j' element of $\xi^{D,k}$. We also use $f(m)$ to denote the tuple (z_m, e_m) where z_m and e_m denote the zone and decision epoch at which the vehicle will become available if it is assigned to path m. Constraints (8) ensure that the each subelement of j'^{th} element of $\xi^{D,k}$ is assigned at most once and Constraints (11) ensure that the number of type i' vehicles assigned is less than the number of vehicles available of type i'.

4.2.3 Benders Decomposition to Efficiently Solve ZACFuture Optimization Formulation

The complexity of the optimization formulation ZACFuture increases with the increase in the number of samples. To reduce this complexity, we exploit the following observation:

Observation 1 *In ZACFuture, once the assignment of vehicles to paths at the current decision epoch (y_{im}) is given, the optimization models for computing the assignment of vehicles to requests at future decision epochs, ($u^k_{ij'r}$) for each of the samples k, are independent of each other.*

The observation 1 allows us to use Benders Decomposition (Benders, 1962) to decompose the large optimization formulation into multiple smaller problems that can be solved in parallel. Benders Decomposition is a master slave decomposition technique where the master problem finds the solutions for the integer variables; and the slave problem(s) is (are) used to find the solutions to all other variables (which can take any value in the interval and need not be integers) while keeping the values of the integer variables fixed to the value obtained by the master problem. The values obtained by slave problems help in generating benders cuts, which are added to the master problem and the master problem is solved again with these cuts to obtain an improved solution. This process is repeated until no more cuts can be added to the master problem. It is widely used to solve such two-stage stochastic problems (Murphy, 2013; Lowalekar et al., 2018).

Based on Observation 1, y_{im} are the difficult variables as they impact the values assigned to all the other variables. x_{jm} are also difficult variables as they can take only integer values. As described in the previous section, the second stage problem for each sample is a weighted bipartite matching problem. The constraint matrix for weighted bipartite matching is totally unimodular, therefore, integrality constraints on the $u^k_{i'j'r}$ variables can be relaxed (Hoffman & Kruskal, 2010) after fixing the values of y_{im} variables. Therefore, the master problem obtains the assignments for the "difficult" integer variables (x_{jm} and y_{im}) and the slave problem(s) obtain the assignments to the $u^k_{ij'r}$ variables.

Master$(\mathcal{P}, Pv, Pr, b, N)$**:**

$$\max \quad \sum_{j \in \mathcal{D}} \sum_{m \in Pr_j} x_{jm} + \frac{1}{|\xi^D|} \sum_{k=0}^{|\xi^D|} \mathcal{Q}(\{y_{im}\}_{i \in \mathcal{V}}, (j \in \mathcal{P}, k) \tag{15}$$

$$s.t. \quad \sum_{m \in Pr_j} x_{jm} \leq 1 ::: \forall j \in \mathcal{D} \tag{16}$$

$$\sum_{m \in Pv_i} y_{im} \leq 1 ::: \forall i \in \mathcal{V} \tag{17}$$

$$\sum_{j \in \mathcal{D}} x_{jm} \cdot b^n_{jm} \leq \sum_i y_{im} \cdot N(i, m, n) ::: \forall m, \forall n \tag{18}$$

Table 5: Optimization Formulation for Master problem - ZACBenders

For the master (Table 5), in the optimization provided in ZACFuture, we replace the part of the objective dealing with future variables, $\{u^k_{i'j'r}\}$ by the recourse function $\mathcal{Q}(\{y_{im}\}_{i \in \mathcal{V},(m \in \mathcal{P}}, k)$, which becomes the objective function in the slave problems. The recourse function $\mathcal{Q}()$ needs to be computed for each value of y_{im}. In the slaves (Table 6), we consider the fixed values of y_{im} and to avoid confusion, we refer to them using the capital letter notation, Y_{im}.

SlavePrimal$(\mathcal{P}, Pv, Pr, b, N, Y, k)$**:**

$$\max \quad \frac{1}{|\xi^D|} \sum_{j' \in \xi^{D,k}} \sum_{r=0}^{\lceil \frac{\eta^k_{j'}}{\kappa} \rceil} \sum_{i'} w^k_{i'j'r} \cdot u^k_{i'j'r} \tag{19}$$

$$s.t. \quad \sum_{j' \in \xi^{D,k}} \sum_{r=0}^{\lceil \frac{\eta^k_{j'}}{\kappa} \rceil} u^k_{i'j'r} \leq \sum_i \sum_m Y_{im} ::: \forall i' \tag{20}$$

$$\sum_{i'} u^k_{i'j'r} \leq 1 ::: \forall j' \in \xi^{D,k}, 0 \leq r < \lceil \frac{\eta^k_{j'}}{\kappa} \rceil \tag{21}$$

Table 6: Optimization Formulation for Slave problem (Primal)- ZACBenders

The dual (Bertsimas & Tsitsiklis, 1997) of the primal slave problems are provided in Table 7, where α variables are the dual variables corresponding to the constraints (20) and β variables are the dual variables corresponding to the constraints (21).

The *weak duality theorem* (Bertsimas & Tsitsiklis, 1997) states that the solution to a maximization primal problem is always less than or equal to the solution of the corresponding dual problem. Therefore, using the concept of weak duality, we can say that, by taking the dual of the slave problems, we can find an upper bound on the value of the recourse function $(\mathcal{Q}())$(objective of primal slave problem), in terms of the master problem variables

SlaveDual($\mathcal{P}, Pv, Pr, b, N, Y, k$):

$$\max \quad \sum_{i'}\sum_{i}\sum_{m\in Pv_i} \alpha^k_{i'} \cdot Y_{im} + \sum_{j'\in \xi^{D,k}}\sum_{r} \beta^k_{j'r} \tag{22}$$

$$s.t. \quad \alpha^k_{i'} + \beta^k_{j'r} \geq w^k_{i'j'r} ::: \forall i', j', r \tag{23}$$

$$\alpha^k_{i'} \geq 0 ::: \forall i', k \tag{24}$$

$$\beta^k_{j'r} \geq 0 ::: \forall j', r, k \tag{25}$$

Table 7: Optimization Formulation for Slave problem (Dual) - ZACBenders

y_{im}. These can then be added as optimality cuts to the master problem (Murphy, 2013) for generating better first stage assignments[11].

Let θ^k be the approximation of $\mathcal{Q}()$ function then the master problem with optimality cuts is provided in the Table 8. It should be noted that we are using y_{im} variables in the "master with optimality cuts" and not the fixed values, Y_{im}.

MasterWithOptimalityCuts($\mathcal{P}, Pv, Pr, b, N$):

$$\max \quad \sum_{j\in\mathcal{D}}\sum_{m\in Pr_j} x_{jm} + \frac{1}{|\xi^D|}\sum_{k}\theta^k \tag{26}$$

$$s.t. \quad \theta^k \leq \sum_{i'}\sum_{i}\sum_{m\in Pv_i} \alpha^k_{i'} \cdot y_{im} + \sum_{j'\in \xi^{D,k}}\sum_{r} \beta^k_{j'r} \tag{27}$$

$$\sum_{m\in Pr_j} x_{jm} \leq 1 ::: \forall j \in \mathcal{D} \tag{28}$$

$$\sum_{m\in Pv_i} y_{im} \leq 1 ::: \forall i \in \mathcal{V} \tag{29}$$

$$\sum_{j\in\mathcal{D}} x_{jm} \cdot b^n_{jm} \leq \sum_{i} y_{im} \cdot N(i,m,n) ::: \forall m, \forall n \tag{30}$$

Table 8: Optimization Formulation for Master problem (with optimality cuts) - ZACBenders

In each iteration we solve the master problem and the computed y_{im} variable values are passed to the dual slave problems. After solving the dual slave problems, optimality cuts are generated. If the current values of $\theta^k(\forall k)$ satisfy the optimality cut conditions, then we have obtained an optimal solution, else cuts are added to the master problem and the master problem is solved again. Figure 9 shows the flow diagram for the same.

The slave problems are independent of each other (Table 7) and are only connected by the choice of the master variables ("difficult" integer variables). Therefore, once the master variables are fixed, the slave problems can be solved in a parallel fashion.

Discussion: The RMP problem defined in Section 2 and proposed ZAC based formulations assumes that requests will not be made in advance. While it is the most common setting in the ridesharing application, it will be useful for customers to have an option to request for

[11] As the slave problems are always feasible for any value of the master variables, we only need to add optimality cuts to the master problem.

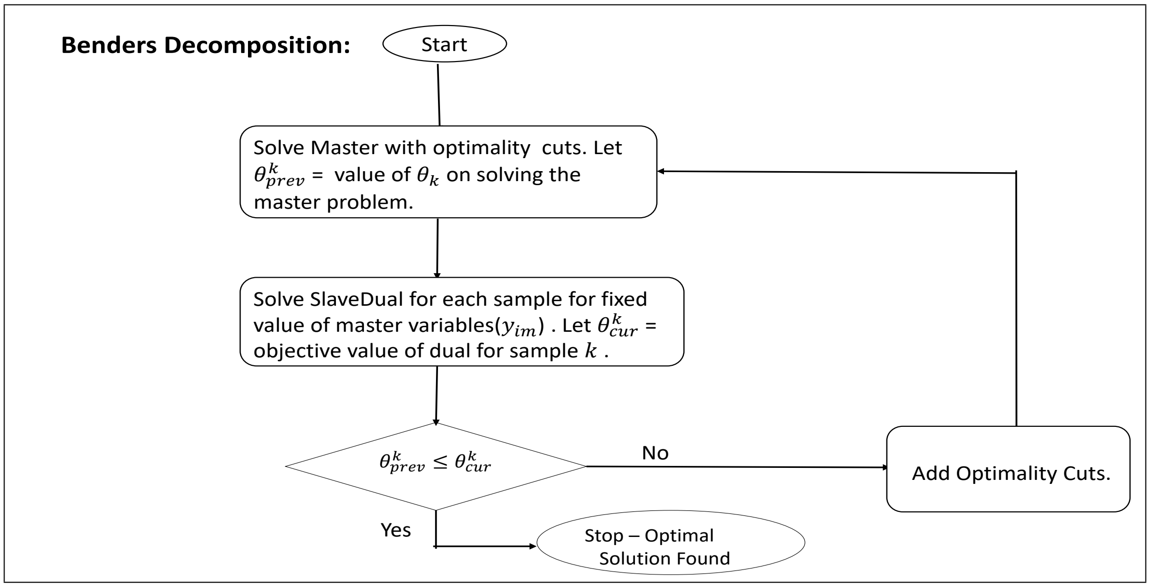

Benders Decomposition:
Start
Solve Master with optimality cuts. Let θ^k_{prev} = value of θ_k on solving the master problem.
Solve SlaveDual for each sample for fixed value of master variables(y_{im}) . Let θ^k_{cur} = objective value of dual for sample k .
$\theta^k_{prev} \leq \theta^k_{cur}$
No
Add Optimality Cuts.
Yes
Stop – Optimal Solution Found

Figure 9: ZACBenders Approach: Finding Optimal Assignment of requests to paths to vehicles

rides in advance. By using ZACBenders, we can allow customers to make advance requests with some limitations. The changes which will be needed in the algorithm are as follows:

1. When requests are made in advance (e.g. two hours in advance), it will be considered as part of each of the sample in ZACBenders.
2. If these advance requests are assigned, in all the future timesteps (until the difference between request pick-up time and current time is less than τ), for these requests, constraints (16) or (21) will be kept as equality constraints instead of inequality constraints , i.e., once accepted, these requests should have a guarantee that a vehicle will arrive on time.

5. Experiments

The goal of the experiments is to evaluate the performance of ZAC and ZACBenders in comparison to TBF [12]. We also compare the performance of our algorithms against NeurADP (Shah et al., 2020), which learns the expected future value of each assignment of vehicle to trips by using a neural network based value function in the approximate dynamic programming framework. As NeurADP requires training a different model for each change in an input parameter, using limited academic resources, it was not possible to run the exhaustive set of experiments with NeurADP. Therefore, we first show the detailed exper-

[12] The complexity of TBF increases with the increase in vehicle capacity. It is not possible to run it up to optimality. Therefore, we run it with the heuristics mentioned in the paper (0.2 second for each vehicle and keeping 30 vehicles for each request (but keeping all request edges)). We use the objective of maximizing the number of requests served for all algorithms. The objective can be changed to the objective of minimizing the delay or maximizing the revenue for both TBF and ZAC.

imental results comparing the performance of TBF, ZAC and ZACBenders and then in Section 5.3.1 we compare against NeurADP. For ZACBenders we kept a maximum time limit of Δ seconds for each assignment, but the Benders Decomposition can converge before the maximum time limit is reached.

We evaluate the algorithms on following metrics: (1) Service Rate, i.e., percentage of total available requests served. (2) Runtime to compute a single step assignment. We experimented by taking demand distribution from two real-world and one synthetic dataset. The details about datasets are provided in Section 5.1.

Table 9 provides the outline for this section. We will show two main results that demonstrate the significant utility of our approaches:

- Our myopic approach ZAC outperforms the current best myopic approach TBF. While the improvement varies, ZAC serves up to 4% more requests on real-world datasets and up to 20% more requests on synthetic dataset.
- Our non-myopic approach ZACBenders further improves the performance of ZAC. It provides 14.7% improvement over TBF. NeurADP when hyper-optimized to the test settings can perform better than ZACBenders on certain cases. However, ZACBenders gets improvements of up to 12.48% when NeurADP is not hyper-optimized for test settings.

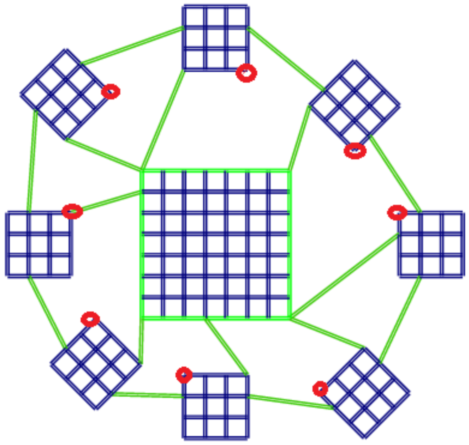
Section	**Description**	**Key Content**
5.1	Datasets	Details on the datasets and different data fields used from the datasets.
5.2	Experimental Settings	Details on the different inputs, parameters and evaluation settings used.
5.3	Results on Real-World Datasets	Describes our key results on real-world datasets and shows the comparison of TBF, ZAC and ZACBenders for different parameters at on the three metrics of service rate and runtime.
5.3.1	Comparison with NeurADP	Describes our key result by comparing TBF, ZAC and ZACBenders with NeurADP.
5.4	Results on Synthetic Dataset	Describes the performance of algorithms on specially created first and last mile scenarios where it is advantageous to explore more request combinations at a decision epoch.

Table 9: Experiment Section Outline

5.1 Datasets

The first real-world dataset is the publicly available New York Yellow Taxi Dataset (NYYellowTaxi, 2016), henceforth referred to as the NYDataset. The name of the other real-world dataset can not be revealed due to confidentiality agreements. It is referred to as Dataset1. We use the street intersections as the set of locations $\mathcal{L}$. To find out the street intersections in real-world dataset, we take the street network of the city from Openstreetmap using osmnx with drive network type (Boeing, 2017). From these we remove the network nodes that do not have any outgoing edges, i.e., we take the largest strongly connected component of the network. For NYDataset, as considered in earlier works by Alonso-Mora et al. (2017a), we only consider the street network of Manhattan as 75% of the requests have pick-up and drop-off locations in Manhattan. Moreover, less than 15% of the total requests have a pick-up and drop-off location in different boroughs of New York indicating that these boroughs can be solved independently.

Both real-world datasets contain data of past customer requests for taxis at different time of the day and for different days of the week. From these datasets, we take the following fields: (1) Pick-up and drop-off locations (latitude and longitude coordinates): These locations are mapped to the nearest street intersection. (2) Pick-up time: This time is converted to appropriate decision epoch based on the value of Δ. The travel time on each road segment of the street network is taken as the daily mean travel time estimate computed using the method proposed by Santi et al. (2014).



(a)

Figure 10: Street network for synthetic dataset. Train stations are marked with red.

To simulate the scenario for on-demand shuttle services (Shotl, 2018; Beeline, 2016; Grab, 2018) having a small set of pick-up/drop-off points in a city, we also perform experiments on a synthetic dataset introduced by Bertsimas et al. (2018). The network (Figure 10) has one downtown area represented by the big square in center and 8 suburbs. We create a train station at one node of each suburb (marked by red circle) to simulate special cases of first and last mile transportation. At each decision epoch, requests are randomly generated by taking pick-up and drop-off location uniformly. In addition, every 180 seconds (frequency of arrival of train at the train stations), we generate first and last mile requests in each suburb (representing arrivals by train).

The number of nodes/locations, edges in the street network of the city and the number of requests present in each dataset are shown in the Table 10.

Dataset	**Locations** ($\|\mathcal{L}\|$)	**Edges** ($\|\mathcal{E}\|$)	**Avg No. of Requests per day (on test days)**	**Avg No. of Requests per hour (Peak) (on test days)**
NYDataset	4373	9540	313683	20910
Dataset1	21212	41424	403770	23664
Synthetic	192	640	173557	8578

Table 10: Details for different datasets

5.2 Experimental Settings

There are three different categories of experimental settings that have an impact on the performance of algorithms:

1. **Inputs provided to all algorithms:** These include

   - **Number of Vehicles ($|\mathcal{V}|$):** The number of vehicles used is dependent on the fleet size of the company. At the start of the experiment, empty vehicles are distributed uniformly at random at different locations. Based on the assignment obtained by algorithms at any decision epoch, the status of vehicles at the next decision epoch is updated. In the results section, we vary the number of vehicles to show the performance of algorithms for different number of vehicles.
   - **Maximum Capacity (κ):** The maximum number of passengers that can be present in a vehicle at any time.
   - **τ and λ:** τ represents the maximum time within which the vehicle should reach the origin location of request and λ denotes the maximum allowed travel delay for any request (in seconds).
   - **Decision epoch duration (Δ):** This parameter determines how often the algorithm should be executed, and assignment decisions are made. For example, if $\Delta = 60$ seconds, then requests are batched for the duration of 60 seconds and the decision of serving or rejecting these requests is taken every 60 seconds by the algorithm. We vary this parameter to show the performance of algorithms for different values.

Input Parameter	**Values considered in Experiments**
Δ (in seconds)	10,30,60
τ (in seconds)	120,180,300,420
λ (in seconds)	240,600,840,900
$\|\mathcal{V}\|$	1000,2000,3000,5000,8000,10000
κ	1,2,3,4,8,10

Table 11: Inputs to all algorithms

Table 11 shows the values of different input parameters considered in the experiments.

2. **Parameters of the algorithm:** The parameters required by our algorithms are:

- **Clustering Method:** To construct zones from the set of locations $\mathcal{L}$, we compare the performance by using different clustering methods and different static zone sizes. Zone size is taken as the intra zone travel time (in seconds).
- **Number of Different Zone Sizes (for drop-off locations) (M):** In the online completion phase of the offline path, instead of using a fixed zone size, ZAC dynamically decides the zone size to be used from a predefined fixed set of zone sizes. We vary the number of different zone sizes from which the ZAC algorithm picks the best zone size for a path.
- **Zone Size for Samples ($\mathscr{Z}^s$):** For samples we use a static zone size of 600 seconds. While it is possible to improve the performance of ZACBenders by using different zone size for different capacities and different value of τ and δ, we observe in the experiments that, by using a fixed zone size, it is possible to get improvement across different parameters and different datasets.
- **Number of Samples ($|\xi^D|$):** While computing an assignment at decision epoch e, our non-myopic approach ZACBenders require samples of customer requests at decision epochs $e+1, e+2, .., e+Q$, where Q = $\lfloor \frac{\rho}{\Delta} \rfloor$, from past data (at the same decision epoch on the past days). We identify the right value for the number of samples through experiments as described in results section. Each sample correspond to past one day. For example, if 10 samples are used, it means that requests from past 10 days are used to compute the expected future value in ZACBenders.
- **Lookahead Duration (ρ):** This determines how far ahead ZACBenders look into the future. If lookahead duration is 600 seconds and current time is 09:00AM, ZACBenders considers samples of customer requests up to 09:10AM.

Table 12 shows the different values for the parameters used in the experiments. To obtain the right set of parameter values, we compare the performance of approaches by running them on 5 different weekdays from 21-03-2016 to 25-03-2016 and taking the average value over these five days.

Algorithm Parameter	**Values considered in Experiments**
M	2,4,6
Clustering Method	GBC, HAC_MAX,HAC_AVG
Zone size for Samples (Z^s) (in seconds)	600
Number of Samples ($\lvert\xi^D\rvert$)	1,3,5,8,10
Lookahead Duration (in seconds) (ρ)	600, 900 ,1200 ,1500

Table 12: Algorithm Parameter Settings

3. **Evaluation settings:**
   - **Evaluation duration:** We evaluate the performance of each algorithm over 1 hour by varying different input and algorithmic parameters. For a subset of parameter combinations, we also compared the performance over 24 hours [13].
   - **Number of days and time of evaluation:** We performed experiments with requests at various times of the day, 8:00 AM, 3:00 PM, 6:00 PM, 12:00AM and on different days. We evaluated the approaches by running them on 15 different weekdays between 04-04-2016 and 22-04-2016 and taking the average values over 15 days. These 15 days are different from the 5 days used to obtain the right set of algorithm parameters.

We conducted experiments with all the combinations of settings and inputs mentioned in this section. To avoid repeating similar results over and over again, we provide the representative results. All experiments are run on a 24 core - 2.4GHz Intel Xeon E5-2650 processor and 256GB RAM. The algorithms are implemented in Java and optimization models are solved using CPLEX 12.6. For ZAC and ZACBenders, we also use offline generated paths. Table 13 provides the number of paths and memory requirement of the offline paths for different values of τ for NYDataset. We generated all possible paths when $\tau \leq 180$ seconds. For higher values of τ, we used the data-driven approach provided in Appendix B.

τ	**Number of Paths**	**Memory used**
120	208604	76 MB
180	2008895	1.68 GB
300	3196478	3.98 GB
420	4256412	7.86 GB

Table 13: Offline Paths

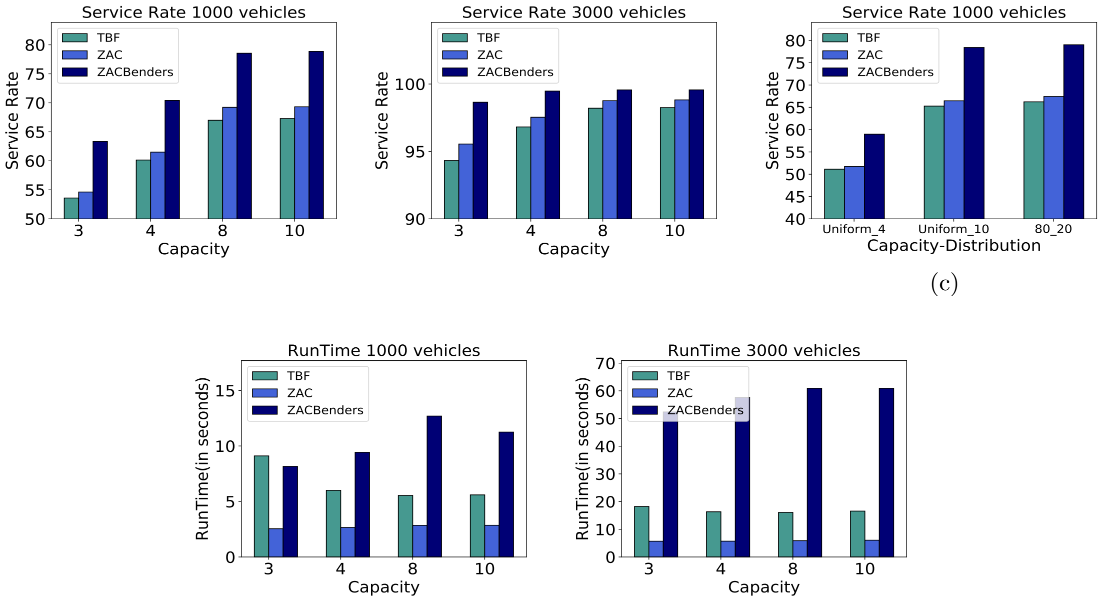
5.3 Results on Real-World Datasets

In this section, we compare the number of requests served by TBF, ZAC and ZACBenders. We also compare the average time taken to compute an assignment by all the approaches.

While we acknowledge that by hyper-optimizing the parameters of our algorithms for each setting, it is possible to improve the results even further, our focus is on showing that even without hyper-optimizing the parameters, our algorithms perform extremely well in comparison to existing methods. Therefore, we choose a single best configuration for parameters of algorithms (justification provided in the Section 5.5). For ZAC and ZACBenders, we cluster locations into zones using HAC_MAX and use M=4 (with zone sizes 0,60,120,300). ZACBenders uses 5 samples with a lookahead duration of 15 minutes and the value of Z^s (zone size used in second stage) is taken as 600 seconds.

[13] As running all the algorithms for 24 hours over different set of parameters takes a long time and the difference in the performance of algorithms over 1 hour was following a similar trend as over 24 hours, we ran it for 24 hours only for a subset of parameters.

We first compare the service rate and runtime of TBF, ZAC and ZACBenders by varying different parameters on two real-world datasets.



(c)

Figure 11: Comparison of ZACBenders, ZAC and TBF on NYDataset for τ=180 seconds, λ=600 seconds and Δ = 60 seconds

Effect of change in vehicle capacity (κ) and number of vehicles ($|\mathcal{V}|$): Figure 11 and Figure 12 show the service rate and runtime comparison of TBF, ZAC and ZACBenders for NYDataset and Dataset1 respectively at 8am (Peak time) .

For the change in the number of vehicles, we make the following observations:

- On Dataset1, the difference in the service rate obtained by ZAC and TBF increases as the number of vehicles increases from 3000 to 5000. One of the reasons is that TBF limits the number of vehicles considered for each request to 30, so the number of requests missed due to this limit will be more for a higher number of vehicles. But on further increasing the number of vehicles to 10000, the gap between ZAC and TBF reduces. This is because, when more vehicles are available, it reduces the need of generating all combinations. On NYDataset, the difference between service rate obtained by ZAC and TBF is maximum for 1000 vehicles.
- The difference in service rate of ZACBenders and ZAC decreases as the number of vehicles are increased. This is because when more vehicles are available, they will be free even after executing current assignments at the current decision epoch, so future demands can be met irrespective of the current assignment. On Dataset1 for capacity 4, ZACBenders obtains 4.2% improvement over ZAC for 1000 vehicles, 3.27% improvement for 3000 vehicles and 2.24% improvement for 5000 vehicles. For 10000 vehicles, the service rate obtained by ZAC and ZACBenders is almost the same. On NYDataset for capacity 4, the maximum improvement obtained by ZACBenders over ZAC is 8.89% which is for 1000 vehicles.

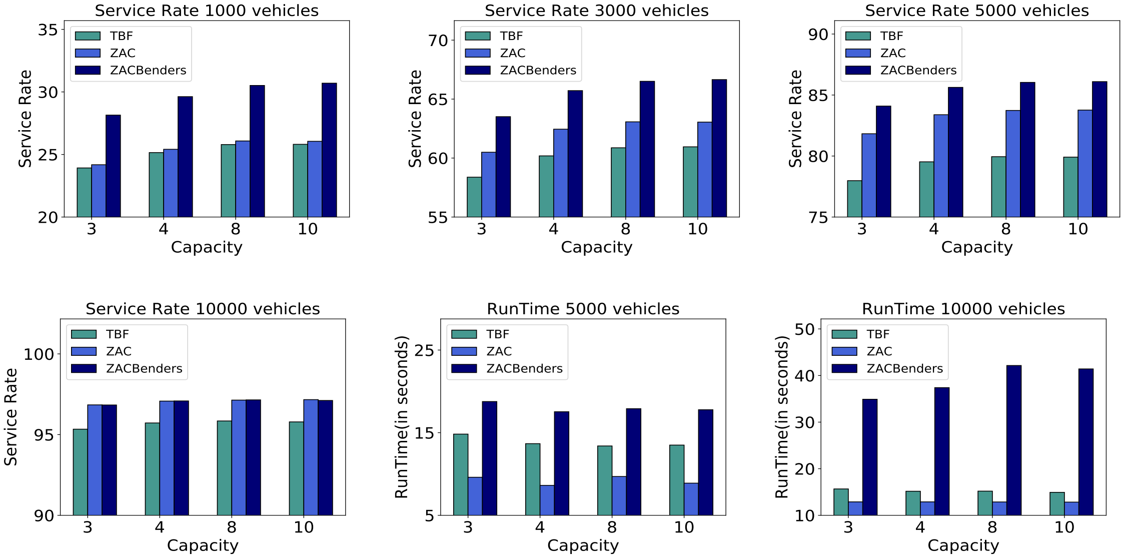


Figure 12: Comparison of ZACBenders, ZAC and TBF on Dataset1 for $\tau = 180$ seconds, $\lambda = 600$ seconds and Δ= 60 seconds

Here are the key observations when vehicle capacity is changed for a fixed number of vehicles:

- Service rate obtained by ZAC is more than TBF for both datasets. For capacity 4 with 1000 vehicles for NYDataset, the service rate obtained by ZAC is 1.36% more than the service rate obtained by TBF and for capacity 10 we obtain a gain of 2.03%. On the other hand, for Dataset1 for capacity 4 with 5000 vehicles, the service rate obtained by ZAC is up to 4% more than the service rate obtained by TBF. On Dataset1, we do not observe much increase in service rate beyond capacity 4 due to the large size of the network and, longer travel times which allow fewer requests to be paired.
- ZACBenders improves the performance of ZAC by using future information. For capacity 4 with 1000 vehicles on NYDataset, it obtains 8.89% improvement over ZAC, which increases to 9.5% for capacity 10. On Dataset1 for 1000 vehicles with capacity 4, ZACBenders obtains 4.2% improvement over ZAC, which increases to 4.6% for capacity 10.
- While both ZAC and TBF can compute a solution in less than 20 seconds, the time taken by ZAC is much less than TBF. The time taken by ZACBenders is much more than the myopic algorithms TBF and ZAC, but the service rate improvement compensates for the additional runtime.

We also experimented with all the vehicles having different maximum capacity. In this case, to compute the weight of edges in bipartite graph in ZACBenders, κ is taken as average of all vehicle's maximum capacity. Figure 11c shows the results where vehicle capacities are generated by taking different distributions. We experimented with the following three distributions:

1. Uniform_4: The maximum capacity of each vehicle is sampled uniformly between 1 to 4.

2. Uniform_10: The maximum capacity of each vehicle is sampled uniformly between 1 to 10.

3. 80_20: 80% of the vehicles have maximum capacity as 4 and 20% of the vehicles have maximum capacity as 6. This is based on the observation that ridesharing companies like Uber, Lyft etc. have a majority of vehicles with maximum capacity 4 and some vehicles with maximum capacity 6.

In this case also, we observe that ZACBenders obtains improvement over myopic approaches. For Uniform_4, the improvement over ZAC is 7.26%, for Uniform_10 the improvement is 11.94% and for 80_20, the improvement obtained is 11.61%. The two-stage stochastic approximation in this case works better than all vehicles having identical maximum capacity as when vehicles have identical maximum capacity, the κ value used in the second stage will be higher and it will allow more requests to group at future decision epoch causing the future value to be overestimated in some cases.

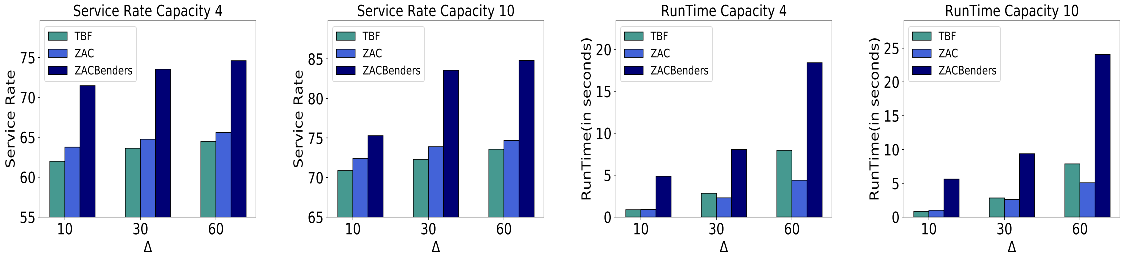


Figure 13: Comparison of ZACBenders, ZAC and TBF for NYDataset for 1000 vehicles and varying values of Δ, $\tau = 300$, $\lambda = 600$ seconds

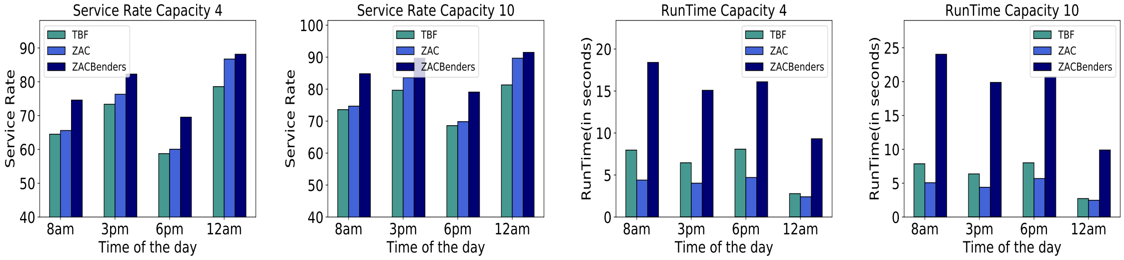


Figure 14: Comparison of ZACBenders, ZAC and TBF for NYDataset for 1000 vehicles and different time of the day. $\tau = 300$, $\lambda = 600$ seconds

Effect of change in value of Δ: We compare the service rate and runtime of algorithms for different values of Δ (Figure 13). Here are the key observations:

- Service rate increases as the value of Δ increases. This is because more requests are available at each decision epoch which allows grouping more requests together.
- The difference between the service rate of ZACBenders and ZAC increases as the Δ increases. One of the reasons is that the value of Δ limits the time available for computation of assignments. When Δ value is low, less number of benders decomposition iterations can be executed within time limit, which affects the performance of ZACBenders.
- The time taken by TBF is much more than ZAC for larger Δ values due to the presence of a higher number of requests at each decision epoch.

Effect of time of the day: We compare the effect of time of day on the performance of algorithms (Figure 14). Here are the key observations:

- The service rate of ZAC is more than TBF in each time interval and ZACBenders further improves this service rate.
- The difference between service rate of ZAC and TBF is more during non-peak hours (3pm and 12am). This is likely as there are less requests available at each decision epoch, so as opposed to peak time where there is more possibility of grouping requests across decision epochs, at non-peak times it is advantageous to explore more combinations at a single decision epoch. The other reason is that ZAC is able to rebalance vehicles better by assigning them to zone paths.

Effect of change in values of τ and λ: We show the service rate and runtime results for different values of τ and λ in Figure 15. Irrespective of the delay constraints, the service rate obtained by ZAC is either more or same as TBF and the runtime of ZAC remains less than TBF in all cases. The improvement in the service rate obtained by ZACBenders over ZAC is also consistent across different values of τ and λ. The time taken by ZACBenders increases as the value of τ and λ increases due to increase in the complexity of the optimization formulation.

On real datasets, ZAC obtains up to 4% gain in service rate over TBF across different parameter values. ZACBenders obtains nearly 10% improvement in service rate over ZAC on NYDataset and 5% improvement on Dataset1 [14]. Typically, even a 0.5% gain is considered significant on real taxi datasets (as shown by a real car aggregation company (Xu et al., 2018)), so the gain obtained by our algorithms is a significant gain.

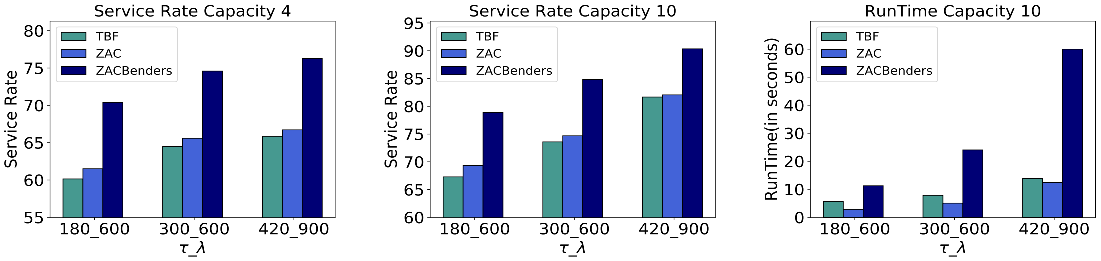
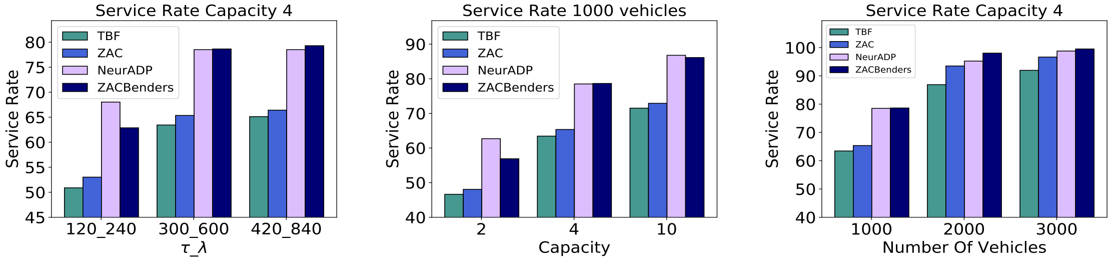


Figure 15: NYDataset - 1000 vehicles, Δ=60 seconds.

[14] This gain further increases when evaluated over longer duration (24 hours) as opposed to 1 hour as seen in the results in the Section 5.3.1.

5.3.1 Comparison with NeurADP

In this section, we compare TBF, ZAC, ZACBenders and NeurADP (Shah et al., 2020) on NYDataset. Both ZACBenders and NeurADP can compute an assignment within maximum Δ seconds. We compare the performance of all algorithms over 24 hours as both ZACBenders and NeurADP consider future information and potentially ignore requests at initial decision epochs to serve more requests in the future and, as a result, achieve higher service rates when evaluated over longer durations. As indicated earlier, NeurADP was specifically trained on these parameter setting for NYDataset. Following are the key observations:



(a) (b) (c)

Figure 16: Comparison of service rate on NYDataset. (a) (b) Number of vehicles =1000 (b)(c) $\tau = 300$,$\lambda = 600$

- For 1000 vehicles of capacity 4 with $\tau = 300$ and $\lambda = 600$ seconds, ZACBenders obtains 14.7% improvement over TBF over 24 hours.
- ZACBenders and NeurADP outperform myopic approaches TBF and ZAC as shown in Figure 16. ZACBenders and NeurADP have comparable performances, except in a couple of cases (with a maximum improvement of 6% for NeurADP).

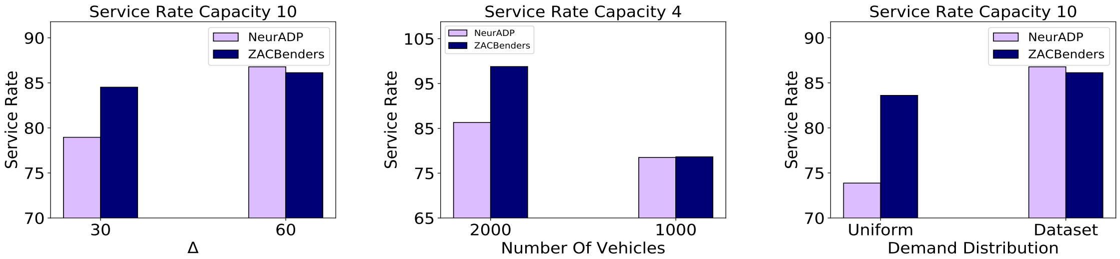
Generalizability : Learning a new model for every possible parameter configuration and every dataset is not scalable nor sustainable for real datasets. For instance, the model training is a time-consuming process and it takes around one week to learn for NeurADP. Unlike NeurADP, ZACBenders does not require training for every parameter setting. Therefore, we now evaluate generalizability of NeurADP in comparison to ZACBenders on test settings across three dimensions:

1. **Change in Decision Epoch Duration (Δ):** If there is a requirement to reduce the decision epoch duration to enhance the user experience, it will not be possible to adopt this change using NeurADP without training a new model and optimizing the hyper parameters efficiently for this case. On the other hand, ZACBenders can be used readily for different values of Δ. As shown in Figure 17a, if we take the model trained for $\Delta = 60$ seconds and use it for $\Delta = 30$ seconds, the performance of NeurADP is significantly affected. In this case, ZACBenders obtains 5.5% improvement in service rate over NeurADP.

2. **Change in Number of Vehicles:** If the ridesharing company decides to increase the fleet size (number of vehicles) and uses the NeurADP model trained for smaller

fleet size, NeurADP is significantly outperformed by ZACBenders. When we take the model trained for 1000 vehicles and use it in a scenario with 2000 vehicles, ZACBenders obtains 12.48% improvement in service rate over NeurADP.

3. **Change in Demand Distribution:** There is going to be an event organized in the city which will cause demand patterns to deviate from the historical patterns. The change in the demand distribution can be predicted and we can get different samples from this predicted demand distribution. To use the NeurADP approach in this scenario, we need to use these samples in the NeurADP simulator and train the neural network model using these samples. So, it will not be possible to use NeurADP immediately. On the other hand, we can provide these samples to ZACBenders and start executing it. In Figure 17c, we show that instead of taking customer requests from the dataset, we evaluate the approaches by taking customer requests from a uniform distribution [15], ZACBenders can obtain 9.73% improvement over NeurADP.



(a) (b) (c)

Figure 17: Comparison of service rate on NYDataset. For these results NeurADP uses the model trained for 1000 vehicles $\Delta = 60$ seconds, $\tau = 300$ seconds, $\lambda = 600$ seconds and the appropriate capacity (4 or 10) (a)(c) Number of vehicles =1000, (a)(b)(c) $\tau = 300$,$\lambda = 600$, (a) (c) Capacity = 10, (b) Capacity = 4

5.4 Results on Synthetic Dataset

The real-world taxi datasets can not capture the scenarios for on-demand shuttle services (Shotl, 2018; Beeline, 2016; Grab, 2018) having a small set of pick-up/drop-off points in a city. These involve scenarios where many requests can be combined at each decision epoch. We represent these scenarios by simulating the case of first and last mile transportation in the synthetic network (details provided in experimental setup), where there are multiple requests at each decision epoch with either identical pick-up location and nearby drop-off locations or identical drop-off locations and nearby pick-up locations resulting in higher possibility of having a large number of request combinations at a decision epoch.

The gain obtained by ZAC over TBF is even more significant in these scenarios as TBF will not be exploring all relevant combinations while ZAC can explore more combinations by using zone paths. ZACBenders provide a slight improvement over ZAC by using future

[15] It can be shown using any distribution, we took uniform distribution as an example.

information but in these scenarios, as the travel times are small and the pick-up and drop-off locations of requests are near each other, the major improvement is obtained by exploring more combinations at a single decision epoch.

We compare the service rate obtained by TBF, ZAC and ZACBenders with different numbers of vehicles and different capacities and make the following observations:

- We observe that with 500 vehicles and capacity 10, ZAC can obtain 20.8% improvement in service rate over TBF. The gain reduces to 16% on increasing vehicles to 1000. This is because when more vehicles are available to serve same number of customer requests, it reduces the need of generating all combinations.
- The service rate obtained by ZACBenders and ZAC is almost the same on this dataset as it is more important to explore more combinations in these scenarios. For 100 vehicles with capacity 10, ZACBenders obtains 2.2% improvement over ZAC and for 500 vehicles with capacity 4 ZACBenders obtains 2% improvement over ZAC.

These results demonstrate that ZAC is able to consider significantly more trips than TBF.

The scope of improvement by incorporating future value is less in this synthetic dataset due to the structure of the network, the nature of requests (mostly first and last mile requests) and lower time windows. As the majority of requests are first and last mile requests with smaller travel times and have a maximum wait time of 2 minutes (and maximum delay of 4 minutes), it makes more sense to group more requests in the same vehicle at the current decision epoch. Considering future information does not help much in this case as most of the requests are first and last mile, so majority of times vehicle remains in the same area (i.e., near train stations). As a result, these vehicles are available to serve new incoming first and last mile requests in-spite of ignoring the future information.

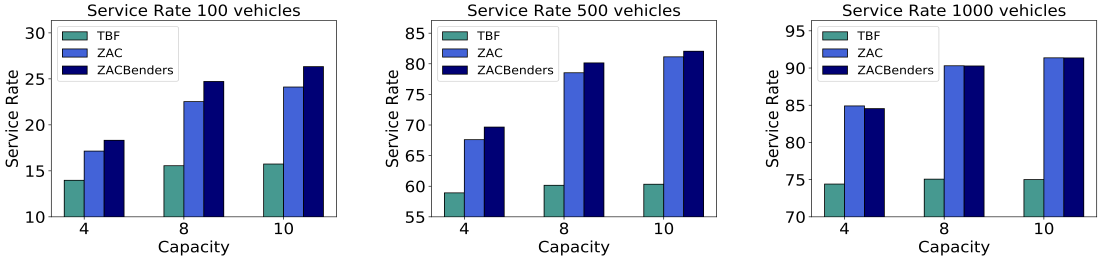
When the future information is ignored, occasionally, vehicles are assigned to requests going towards the downtown area, which have higher travel time. This causes vehicles to move away from train stations and makes them unavailable for serving future first and last mile requests. By considering the future information, we can ignore the requests going into the downtown and keep the vehicle near train stations to serve more first and last mile requests in the same time it will serve one request to the downtown area. As these occurrences are rare (due to the majority of requests being first and last mile), we obtain a small improvement by considering future information in this case.

5.5 Justification for Values of Algorithmic Parameter Settings

In this section, we show the reason for using the fixed algorithmic parameter values (used in previous sections) for ZAC and ZACBenders.

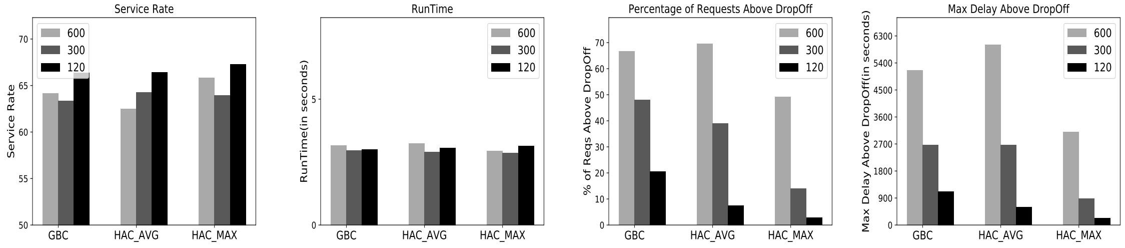
5.5.1 Identification of Right Clustering Method

We first conduct experiments by using different clustering methods, with $M = 1$, by varying the zone sizes. Zone size is taken as the intra zone travel time (in seconds). Figure 19 shows the comparison of GBC, HAC_MAX and HAC_AVG on NYDataset. We compare the service rate, runtime and abstraction error with different clustering methods and different zone sizes for ZAC. We measure abstraction error by computing the percentage of requests



(a)

Figure 18: Synthetic Dataset with $\tau = 120$,$\lambda = 240$ and $\Delta = 60$ seconds



(a) (b) (c) (d)

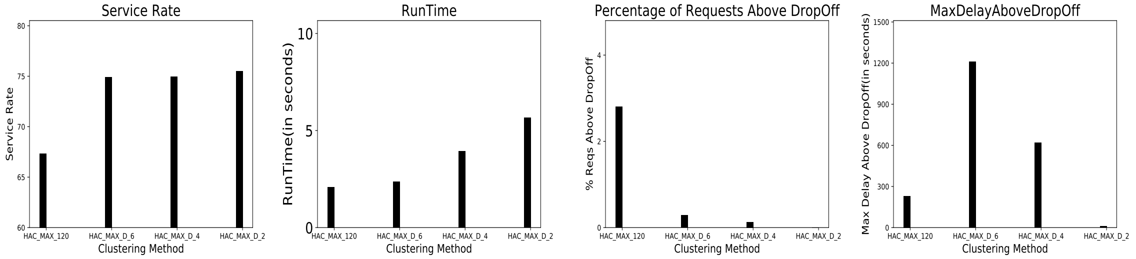
Figure 19: Comparison of service rate, runtime and abstraction error with different clustering methods and zone sizes for $M = 1$, number of vehicles = 1000, capacity = 10, τ =300, $\lambda = 600$ seconds

having delay above λ and maximum delay obtained by any request that is above λ. We can observe that with HAC_MAX more requests can be served while keeping the error due to abstraction minimal. We also observe that as the zone size decreases, the number of requests served increases, error due to abstraction decreases with a slight increase in runtime. Based on these results, we use HAC_MAX as the clustering method for our next set of experiments.

5.5.2 Identification of Right Value of M

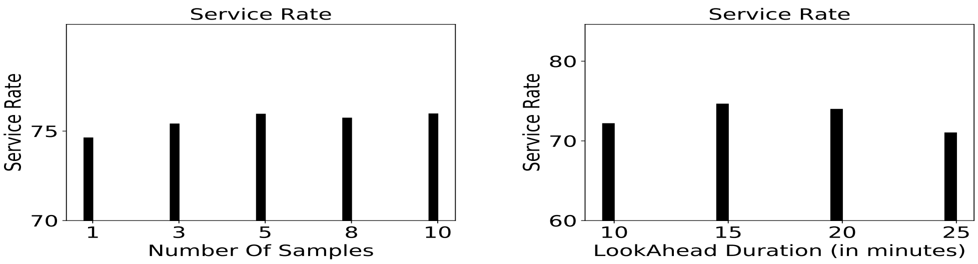
Our next set of experiments compare the service rate, runtime and abstraction error obtained using different values of M. Based on the observations made earlier, for $M = 1$, we use HAC_MAX with zone size 120. For $M > 1$, the clustering method used is HAC_MAX and we run the experiments with different values of M. We use the zone sizes as 0, 60, 120, 300, 480, 600. The zone size of 0 means that the actual locations in the street network are used. Zone size of 60 means that the intra zone travel time is 60 seconds and so on. For $M = 2$, zone sizes used are 0 and 60, for $M = 4$ zone sizes used are 0, 60, 120 and 300 and for $M = 6$, zone sizes used are 0, 60, 120, 300, 480, 600.

We show the comparison of service rate and runtime with $M = 1$ (with zone size 120) and different values of M in Figure 20. HAC_MAX_120 is used to denote that $M = 1$ with



(a) (b) (c) (d)

Figure 20: Comparison of service rate, runtime and abstraction error with different values of M and zone sizes for NYDataset, number of vehicles = 1000, capacity 10, τ = 300, λ = 600 seconds



(a) (b)

Figure 21: Comparison of service rate for different number of samples and lookahead duration number of vehicles = 1000, capacity = 4, τ =300, λ = 600 seconds

zone size 120 is used. HAC_MAX_D_$< m >$ denotes that value of M used is m. From the Figure 20 we can observe that we can serve more requests when $M > 1$, as compared to using fix large size zones. The abstraction error also reduces significantly by using $M > 1$. As the value of M is reduced, quality of the solution improves with the increase in runtime. With $M = 2$ (for zone sizes 0 and 60) , the abstraction error is almost 0 but runtime also increases. With $M = 4$, the abstraction error is less than 1%.

From these experiments on the NYDataset, we obtain that by clustering locations into zones using HAC_MAX and using M=4 (with zone sizes 0,60,120,300), we get the right trade-off between computational complexity and solution quality. Therefore, we use this configuration for ZAC and ZACBenders. We now identify the right number of samples and lookahead duration for the ZACBenders algorithm.

5.5.3 Number of Samples

We compare the service rate by varying number of samples from 1 to 10 as shown in Figure 21a. We can observe from the figure, the service rate obtained by using a single sample is 74.6% and increases to 75.9% on using 5 samples. The service rate obtained by using 10 samples is 75.96% which is only 0.06% more than the service rate obtained by 5 samples.

As the improvement beyond 5 samples is not much and comes at the cost of extra average runtime, we use 5 samples for the ZACBenders algorithm.

5.5.4 LookAhead Duration

We compare the service rate by varying the lookahead duration from 10 minutes to 25 minutes for a single sample as shown in Figure 21b. We can observe from the figure that the service rate obtained by using lookahead duration of 10 minutes is 72.16% and increases to 74.6% on using lookahead of 15 minutes. The service rate obtained by using higher lookahead is less as with higher lookahead more requests are present at future decision epochs which increases the complexity of the problem and so the number of benders decomposition iterations that can be completed within the maximum time limit reduces, affecting the performance of ZACBenders. Moreover, due to the approximations used in the ZACBenders algorithm, it is not necessary that the performance will improve on using higher lookahead duration. Based on these experimental results, we choose a lookahead of 15 minutes for the ZACBenders algorithm.

6. Related Work

Given the practical and environmental benefits of ridesharing systems, there has always been a lot of interest in developing algorithms for performing matching in these systems. The ridesharing problem is related to the Online Multi-Vehicle Pick-up and Delivery problems which typically represent problems where there are multi-capacity vehicles that transport multiple resources/loads from their origins to destinations. When vehicles are used to move people instead of resources, the problem is referred to as dial-a-ride problem (Feuerstein & Stougie, 2001; Lipmann et al., 2002; Bonifaci et al., 2006) and when all the origins or all the destinations are located at a depot, the problem is referred to as vehicle routing problem (Ritzinger et al., 2016). The general representation of the dial-a-ride problems is ideally suited to represent problems faced by companies such as Super Shuttle (transports people from an airport to different locations in the city), Uber Pooling (transports customers from near by start locations to near by destination locations). But these problems are hard to solve and the traditional approaches for these problems can solve only very small instances of 96 requests and 8 vehicles (Ropke et al., 2007).

The integer programming formulation, without any spatial or temporal aggregation (Ropke et al., 2007), is difficult to solve and is not scalable to large scale problems and online decision-making even for unit-capacity. Therefore, in recent times, many heuristic approaches have been proposed to solve the real-time taxi ridesharing problem. As shown in Figure 1, the existing work on ridesharing systems can be categorized along three dimensions of capacity, sequential or batch consideration of requests and the nature of assignment (whether it is myopic or takes future demand into account for making current assignments).

In case of unit-capacity ridesharing systems, vehicles need to be assigned to at most one request at a time. Greedy and randomized ranking (Karp et al., 1990) algorithms have also been used in the literature to compute myopic matching when requests are considered sequentially. The myopic matching for the batch case is also trivial in this case and can be achieved by performing a bipartite matching between vehicles and requests (Agatz et al., 2011). To improve the performance of these myopic algorithms in the batch case, there has been research on providing approximate dynamic programming (Simao et al., 2009), reinforcement learning (Xu et al., 2018; Lin et al., 2018) and multi-stage stochastic optimization approaches (Lowalekar et al., 2018) which consider multiple samples of future demand to estimate the expected future value of current assignments. While we also consider the batch case and use stochastic optimization approaches, our work is different from this thread as we focus on high capacity ridesharing, where computing a myopic assignment in itself is a challenging problem. To further include the future demand samples and solve the optimization in real-time, we need to propose multiple approximations.

For multi-capacity ridesharing, due to the complexity of finding a myopic batch assignment, most of the existing works consider sequential (i.e., one by one) assignment. Widdows et al. (2017) and Tang et al. (2017) propose an approach which allows 2 passengers to travel in the vehicle at the same time. It takes one request at a time and generates all feasible driver paths by inserting the pick-up and drop-off of the request in the existing driver paths. Pelzer et al. (2015) also allow 2 passengers to share the ride. They divide the road network into multiple partitions and limit the search space within the partition to find the match for the incoming request. Ma et al. (2013) propose a myopic sequential matching algorithm for high capacity ridesharing. They propose a taxi searching algorithm which uses a spatio-temporal index to quickly retrieve candidate taxis. It then uses a scheduling algorithm to find the appropriate taxi for the request. The scheduling algorithm compares the additional incurred distance on inserting the current request in the schedule of each candidate taxi. The taxi which minimizes the additional incurred distance for the request is identified as the appropriate taxi for the request. Other works (Huang et al., 2014; Tong et al., 2018; Chen et al., 2018; Cheng et al., 2017) also provide approaches where insertion operation is widely utilized, i.e., for each request, they find the best place to insert in a taxi's path.

While the sequential solution is faster to compute, the quality of solution obtained is typically poor, therefore, there have been works on finding a myopic batch solution. Most of the works in this case have focussed on low capacity vehicles. Zheng et al. (2018) consider batch assignment but they only consider grouping at most two requests in a vehicle. They propose a different approximation and apply matching and optimization based approaches to assign vehicles to the combination of two requests. Dutta (Dutta, 2018) use a locally sensitive hashing technique to efficiently group two requests together but they do not consider assignment of vehicles to requests. Browns (2016) propose exhaustively generating the combinations of at most three requests from all the available requests. For capacity two vehicles, Yu et al. (2019) propose an approximate dynamic programming approach which is non-myopic. They use a linear value function approximation to approximate the future value of assignment and use spatial and temporal aggregation to group different parts of road network into a small number of regions. Their approach is not scalable to a large number of locations and higher capacity vehicles.

A leading approach for high capacity ridesharing was provided by Alonso-Mora et al. (2017a). This is a myopic batch assignment approach which as discussed in Section 1 employs different heuristics for online execution. The existing non-myopic batch assignment approaches for high-capacity ridesharing (Alonso-Mora et al., 2017b; Shah et al., 2020) use the myopic approach by Alonso-Mora et al. (2017a) as a base approach. These approaches have drawbacks where either they can not be executed in real-time or can not be easily adapted to different settings and parameters.

The non-myopic approach by Alonso-Mora et al. (2017b) is a minor extension of their myopic approach where they randomly sample 200 or 400 requests for next 30 minutes and then use those requests along with currently available requests to generate the assignments. Typically, there are 300 requests per minute so randomly sampling 200 or 400 requests for the next 30 minutes does not help in improving the quality of solution. This is reflected in their results as well, where the service rate remains approximately same as the service rate of the myopic approach. They observe a minor decrease in the average delay experienced by the passengers. The sampled requests also increase the computational complexity of the approach and the runtime of the approach after adding sampled requests is more than the time available for assignment(duration over which requests are batched). As a result, it is not possible to use the approach for real-time assignments. The non-myopic approach by Shah et al. (2020) uses a similar approach as Alonso-Mora et al. (2017a) to generate the feasible trips and then use a neural network based value function approximation to estimate the future value of the current assignment of vehicle to trips. While the approach greatly outperforms the myopic approaches, due to the need of training a separate network model for each dataset and each change of input parameter, it is not easily adaptable to different settings as shown in our experimental results.

The zone path construction based approaches proposed in this work, overcome these limitations of existing work by providing an offline-online method to generate request combinations efficiently by employing zone-paths. The future value of assignment to these zone paths is computed by considering multiple samples of future demand.

7. Conclusion

In this paper, we presented zone path construction based approaches that can efficiently perform ridesharing for higher capacity vehicles. The experimental comparison on real-world and synthetic datasets show that our approach can outperform the current best myopic approach (used even by taxi and car aggregation companies like Grab and Lyft) in terms of both runtime and solution quality. Our non-myopic approach further improves the performance by using multiple future demand samples and outperforms the state-of-the-art approaches.

8. Acknowledgements

This work was partially supported by the Singapore National Research Foundation through the Singapore-MIT Alliance for Research and Technology (SMART) Centre for Future Urban Mobility (FM). We thank Sanket Shah for providing valuable comments that greatly improved the paper.

Appendix A. Zone Creation

We cluster the set of locations $\mathcal{L}$ into zones. In this work, we mainly explored the following methods to cluster locations into zones:

1. *Grid Based Clustering (GBC):* As shown in Figure 22a, the city can be divided into different parts using square grid cells. Each square grid represents a zone. The size of

square grid can be used to determine the time taken to move from one zone to another zone or within a zone [16]. It provides a simple method to divide city of any size into multiple zones. A very similar grid based clustering is used by Ma et al. (2013).

2. *Hierarchical Agglomerative Clustering (HAC):* We also employed Hierarchical Agglomerative clustering (Rocach & Maimon, 2005) to cluster locations into zones (Figure 22b). The algorithm starts by including each location in a different cluster/zone and the distance between two locations is measured in terms of the time taken to travel between them. At each iteration of the algorithm, the two closest clusters are merged to form a single cluster. The process continues until the minimum inter-cluster travel time between any two clusters is more than a given threshold. We define the inter cluster travel time using the following two linkage criterion:
(1) *Complete Linkage (HAC_MAX):* The time taken to travel between two clusters is the maximum time required to travel between two locations of different clusters, i.e.,

$$T(X,Y) = \max_{x \in X, y \in Y} t(x,y)$$

where X and Y denote the two clusters and x and y denote the locations. $t(x,y)$ denotes the time taken to travel from location x to y and $T(X,Y)$ denotes the time taken to travel from cluster X to cluster Y. Complete linkage tends to find compact clusters of approximately equal diameters.
(2) *Mean Linkage (HAC_AVG):* The time taken to travel between two clusters is the average of the time required to travel between any two locations of different clusters, i.e.,

$$T(X,Y) = \frac{1}{|X||Y|} \sum_{x \in X,} \sum_{y \in Y} t(x,y)$$

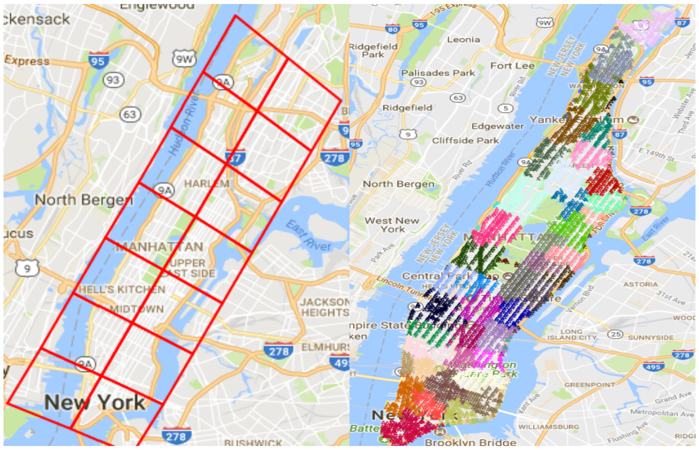
As mentioned before, we use these methods as they do not require prior knowledge about the number of clusters and have been used in earlier works on similar problems (Ma et al., 2013; Hasan et al., 2018). We perform experiments comparing these different zone creation methods and use the method which provides right trade-off between computational complexity and solution quality. (Please refer to Section 5.5 for detailed results).

Appendix B. Offline Partial Path Generation: Data-Driven Approach

While for smaller networks and for smaller values of τ (≤ 180 seconds) for larger networks, we can generate and store all possible paths, but due to the exponential increase in the number of paths with increasing value of τ for large networks, it is necessary to limit the number of paths. One possible way to reduce the number of paths is by using zones instead of locations in the offline path generation. However, this can lead to additional delay during pick-up, which is not preferred. Therefore, we use the following data-driven approach to reduce the number of paths for larger τ values.

The key idea in the data-driven approach is to only keep the paths that have a high likelihood of serving large number of requests with the assumption that future requests will follow the historical demand pattern.

[16] Presence of one way streets can make the computation of actual time taken a bit challenging.



(a) (b)

Figure 22: Abstraction of Locations into Zones (a) Grid based clustering (b) Clusters formed by Hierarchical Clustering

When the value of τ is large, we break τ into multiple smaller time intervals $\tau_1, \tau_2, ...\tau_n$, such that $\tau = \sum_i \tau_i$. The value $\tau_i, \forall i$ is taken such that we can generate and store all possible paths of τ_i duration.

We process all possible paths for τ_1, τ_2,..,τ_n duration against the requests available in historical data. We ignore the paths for which the average number of requests served per minute is less than a threshold γ_i (learned experimentally) and then combine only those paths for which number of requests served is greater than γ_i. This process ensures that we keep the paths which are more likely to serve high number of requests.

By following the above process, there is a chance that no path gets recorded between two locations that can be reached in τ seconds. For such location pairs, we keep the shortest path to travel between them.

As an example, in our experiments, for $\tau = 300$ seconds, we consider $\tau = \tau_1 + \tau_2$, where $\tau_1 = 120$ seconds and $\tau_2 = 180$ seconds and we take γ_1 and γ_2 as 1. The paths of τ_1 and τ_2 duration are processed against 20 days of historical data.

Appendix C. Pseudocode for GetPathsForVehicle Function Used in ProcessOfflinePartialPaths

In this section, we describe the pseudocode for GetPathsForVehicle function used in the Algorithm 3 in main paper. In step 12, we store the destination location of the requests previously assigned to vehicles. A vehicle should deviate from its current path only if it can be assigned to a new request, therefore, in step 10, we consider only those paths which can pick at least one of the newly available requests. Steps 20-22 ensure that we consider only those paths which can potentially satisfy all the previously assigned requests for a vehicle. This is because a vehicle will be assigned to a path if and only if it can serve all previously assigned requests.

Algorithm 5 GetPathsForVehicle($i, q_i, \mathcal{R}[k], \mathcal{R}_p[k], \mathcal{P}_{off}$)

```
$plist = \{\}, plist[i] = 0, \forall i$
$\mathcal{R}^i$=[],$\mathcal{R}^i_p$=[]
for $r \in q_i$ do
    if $r$ is already picked then
        $\mathcal{P}^{h,r}_{off} = GetPathsFromIndex(\mathcal{P}^{h}_{off}, \nu_i, \omega_i, \omega_i)$
    else
        $\mathcal{P}^{h,r}_{off} = GetPathsFromIndex(\mathcal{P}^{h}_{off}, o_r, a_r - t, a_r - t + \tau)$
    for each path $k \in \mathcal{P}^{h,r}_{off}$ do
        $plist[k] += 1$
        if $|\mathcal{R}[k]| > 0$ then
            $lb_j = a_j - t + \mathcal{T}(o_j, d_j), ub_j = lb_j + \lambda$
            $\mathcal{R}^i[k].add(d_j, lb_j, ub_j)$
            $\mathcal{R}^i_p[k].add(o_j)$
            if $lb_j < \tau$ then
                if $k$ visits $d_j$ then
                    $\mathcal{R}_p[k].add(d_j)$
                else if $ub_j < \tau$ then
                    $\mathcal{R}[k].remove(d_j, (lb_j, ub_j))$
for each path $k \in$ plist do
    if $|plist[k]| == |q_i|$ then
        $\mathcal{R}[k].addAll(\mathcal{R}^i[k])$
        $\mathcal{R}_p[k].addAll(\mathcal{R}^i_p[k])$
return $\mathcal{R}[k], \mathcal{R}_p[k]$
```

Appendix D. Rebalancing of Unassigned Vehicles

Similar to Alonso-Mora et al. (2017a), we perform a rebalancing of unassigned vehicles to high demand areas by using the same method as them. We make similar assumptions

1. Unserved customers may request again.

2. At future timesteps, more customer requests may originate from the areas where we could not serve requests at current timestep.

Therefore, to rebalance the unassigned vehicles, after each batch assignment, the unassigned vehicles are assigned to unserved requests to minimize the sum of travel times, with the constraint that either all unserved requests or all of the unassigned vehicles are assigned. The linear program is provided in table 14. Let $\mathcal{V}_u$ denotes the set of unassigned vehicles and $\mathcal{D}_u$ denote the set of unserved customer requests. m_{ij} is a binary variable indicating that vehicle i is moving towards customer request j.

Appendix E. Complexity Analysis for Offline-Online Generation of Zone Paths

We analyse the computational complexity of the Offline-Online Generation of Zone Paths. We look at the complexity of each step.

RebalanceVehicles():

$$\min \sum_{j \in \mathcal{D}_u} \sum_{i \in \mathcal{V}_u} \mathcal{T}(\nu_i, o_j) * m_{ij} \tag{31}$$

$$\textit{subject to} \quad \sum_{i \in \mathcal{V}_u} m_{ij} \leq 1 ::: \forall j \in \mathcal{D}_u \tag{32}$$

$$\sum_{j \in \mathcal{D}_u} m_{ij} \leq 1 ::: \forall i \in \mathcal{V}_u \tag{33}$$

$$\sum_{j \in \mathcal{D}_u} \sum_{i \in \mathcal{V}_u} m_{ij} = min(|\mathcal{V}_u|, |\mathcal{D}_u|) \tag{34}$$

$$0 \leq m_{ij} \leq 1 ::: \forall i, j \tag{35}$$

Table 14: Optimization Formulation for Rebalancing unassigned vehicles

1. **Offline Partial Path Generation:** This step requires generating all paths in the network of $\mathcal{G}$ duration τ. The network $\mathcal{G}$ has $\mathcal{L}$ nodes and $\mathcal{E}$ edges. As the graph is not fully connected, we assume a branching factor of b (average number of neighbors of any vertex $l \in \mathcal{L}$). Let the maximum length of a duration τ path is k. Therefore, the complexity of this step is the complexity of generating all paths of length k in a network with $\mathcal{L}$ nodes, $\mathcal{E}$ edges and branching factor b. The total possible number of paths is $|\mathcal{L}| * b^k$. The total nodes explored in path generation are $|\mathcal{L}| * (1 + b^1 + b^2 + ... + b^k)$ $= |\mathcal{L}| * \frac{b^{k+1} - 1}{b - 1}$.

2. **Online Processing of Offline Partial Paths:** The offline partial paths are processed using the current demand $\mathcal{D}$. Suppose α denotes the fraction of paths in which the pick-up location of any demand element is present. Then, the total complexity of online processing of offline partial paths is $O(|\mathcal{D}| * \alpha * |\mathcal{L}| * b^k)$.

3. **Online Completion of offline partial paths:** Let the total number of unique partial paths after online processing step is $\omega * |\mathcal{L}| * b^k$. Now suppose each path is associated with maximum β drop-off locations after considering the appropriate zone size out of the M available zone sizes. As we need to perform exhaustive search on these β locations, the complexity of this step is $O((\beta!) * \omega * |\mathcal{L}| * b^k)$.

As the network $\mathcal{G}$ is generally very sparse and each of the individual steps can be parallelized, the actual computation time is much less allowing to execute this in real-time.

References

Agatz, N., Erera, A. L., Savelsbergh, M. W., & Wang, X. (2011). Dynamic ride-sharing: A simulation study in metro atlanta. *Procedia-Social and Behavioral Sciences*, *17*, 532–550.

Alonso-Mora, J., Samaranayake, S., Wallar, A., Frazzoli, E., & Rus, D. (2017a). On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proceedings of the National Academy of Sciences*, 201611675.

Alonso-Mora, J., Wallar, A., & Rus, D. (2017b). Predictive routing for autonomous mobility-on-demand systems with ride-sharing. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3583–3590. IEEE.

An, L., Machra, M., Moser, A. M., Simonovikj, S., & Larson, R. C. (2019). Queues in service systems: Some unusual applications. In *Handbook of Service Science, Volume II*, pp. 327–348. Springer.

Beeline (2016). Beeline singapore - book seat on the buses. https://www.beeline.sg/.

Benders, J. F. (1962). Partitioning procedures for solving mixed-variables programming problems. *Numerische mathematik*, *4*(1), 238–252.

Bertsimas, D., Jaillet, P., & Martin, S. (2018). Online vehicle routing: The edge of optimization in large-scale applications. *Operations Research*.

Bertsimas, D., & Tsitsiklis, J. N. (1997). *Introduction to linear optimization*, Vol. 6. Athena Scientific Belmont, MA.

Boeing, G. (2017). Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, *65*, 126–139.

Bonifaci, V., Lipmann, M., Stougie, L., et al. (2006). *Online multi-server dial-a-ride problems*. TU/e, Eindhoven University of Technology, Department of Mathematics and . . . .

Browns, T. (2016). Match making in lyft line. https://eng.lyft.com/matchmaking-in-lyft-line-9c2635fe62c4.

Chen, L., Gao, Y., Liu, Z., Xiao, X., Jensen, C. S., & Zhu, Y. (2018). Ptrider: a price-and-time-aware ridesharing system. *Proceedings of the VLDB Endowment*, *11*(12), 1938–1941.

Cheng, P., Xin, H., & Chen, L. (2017). Utility-aware ridesharing on road networks. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1197–1210. ACM.

Dube-Rioux, L., Schmitt, B. H., & Leclerc, F. (1989). Consumers' reactions to waiting: when delays affect the perception of service quality. *ACR North American Advances*.

Dutta, C. (2018). When hashing met matching: Efficient search for potential matches in ride sharing. *arXiv preprint arXiv:1809.02680*.

Feuerstein, E., & Stougie, L. (2001). On-line single-server dial-a-ride problems. *Theoretical Computer Science*, *268*(1), 91–105.

Grab (2018). Grab shuttle plus - on demand shuttle service. https://www.grab.com/sg/shuttleplus/.

Hasan, M. H., Van Hentenryck, P., Budak, C., Chen, J., & Chaudhry, C. (2018). Community-based trip sharing for urban commuting..

Hoffman, A., & Kruskal, J. (2010). Introduction to integral boundary points of convex polyhedra. *Jünger M et al (eds)*, *50*, 1958–2008.

Huang, Y., Bastani, F., Jin, R., & Wang, X. S. (2014). Large scale real-time ridesharing with service guarantee on road networks. *Proceedings of the VLDB Endowment*, *7*(14), 2017–2028.

Karp, R. M., Vazirani, U. V., & Vazirani, V. V. (1990). An optimal algorithm for on-line bipartite matching. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pp. 352–358. ACM.

Lin, K., Zhao, R., Xu, Z., & Zhou, J. (2018). Efficient large-scale fleet management via multi-agent deep reinforcement learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &#38; Data Mining*, pp. 1774–1783. ACM.

Lipmann, M., Lu, X., de Paepe, W. E., Sitters, R. A., & Stougie, L. (2002). On-line dial-a-ride problems under a restricted information model. In *European Symposium on Algorithms*, pp. 674–685. Springer.

Lowalekar, M., Varakantham, P., & Jaillet, P. (2018). Online spatio-temporal matching in stochastic and dynamic domains. *Artificial Intelligence*, *261*, 71–112.

Ma, S., Zheng, Y., & Wolfson, O. (2013). T-share: A large-scale dynamic taxi ridesharing service. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pp. 410–421. IEEE.

Maister, D. H., et al. (1984). *The psychology of waiting lines*. Harvard Business School Boston, MA.

Murphy, J. (2013). Benders, nested benders and stochastic programming: An intuitive introduction. *arXiv preprint arXiv:1312.3158*.

NYYellowTaxi (2016). New york yellow taxi dataset. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

Parragh, S. N., Doerner, K. F., & Hartl, R. F. (2008). A survey on pickup and delivery problems. *Journal für Betriebswirtschaft*, *58*(1), 21–51.

Pelzer, D., Xiao, J., Zehe, D., Lees, M. H., Knoll, A. C., & Aydt, H. (2015). A partition-based match making algorithm for dynamic ridesharing. *IEEE Transactions on Intelligent Transportation Systems*, *16*(5), 2587–2598.

Ritzinger, U., Puchinger, J., & Hartl, R. F. (2016). A survey on dynamic and stochastic vehicle routing problems. *International Journal of Production Research*, *54*(1), 215–231.

Rocach, L., & Maimon, O. (2005). Clustering methods data mining and knowledge discovery handbook. *Springer US*, 321.

Ropke, S., & Cordeau, J.-F. (2009). Branch and cut and price for the pickup and delivery problem with time windows. *Transportation Science*, *43*(3), 267–286.

Ropke, S., Cordeau, J.-F., & Laporte, G. (2007). Models and branch-and-cut algorithms for pickup and delivery problems with time windows. *Networks*, *49*(4), 258–272.

Santi, P., Resta, G., Szell, M., Sobolevsky, S., Strogatz, S. H., & Ratti, C. (2014). Quantifying the benefits of vehicle pooling with shareability networks. *Proceedings of the National Academy of Sciences*, *111*(37), 13290–13294.

Shah, S., Lowalekar, M., & Varakantham, P. (2020). Neural approximate dynamic programming for on-demand ride-pooling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34, pp. 507–515.

Shotl (2018). Shotl on demand shuttles. https://shotl.com/.

Simao, H. P., Day, J., George, A. P., Gifford, T., Nienow, J., & Powell, W. B. (2009). An approximate dynamic programming algorithm for large-scale fleet management: A case application. *Transportation Science*, *43*(2), 178–197.

Tang, M., Ow, S., Chen, W., Cao, Y., Lye, K.-w., & Pan, Y. (2017). The data and science behind grabshare carpooling. In *Data Science and Advanced Analytics (DSAA), 2017 IEEE International Conference on*, pp. 405–411. IEEE.

Tong, Y., Zeng, Y., Zhou, Z., Chen, L., Ye, J., & Xu, K. (2018). A unified approach to route planning for shared mobility. *Proceedings of the VLDB Endowment*, *11*(11), 1633–1646.

Widdows, D., Lucas, J., Tang, M., & Wu, W. (2017). Grabshare: The construction of a realtime ridesharing service. In *Intelligent Transportation Engineering (ICITE), 2017 2nd IEEE International Conference on*, pp. 138–143. IEEE.

Xu, Z., Li, Z., Guan, Q., Zhang, D., Li, Q., Nan, J., Liu, C., Bian, W., & Ye, J. (2018). Large-scale order dispatch in on-demand ride-hailing platforms: A learning and planning approach. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 905–913. ACM.

Yang, J., Jaillet, P., & Mahmassani, H. (2004). Real-time multivehicle truckload pickup and delivery problems. *Transportation Science*, *38*(2), 135–148.

Yu, X., & Shen, S. (2019). An integrated decomposition and approximate dynamic programming approach for on-demand ride pooling. *IEEE Transactions on Intelligent Transportation Systems*.

Zheng, L., Chen, L., & Ye, J. (2018). Order dispatch in price-aware ridesharing. *Proceedings of the VLDB Endowment*, *11*(8), 853–865.