Optimal Online Dispatch for High-Capacity Shared Autonomous Mobility-on-Demand Systems

Cheng Li[1,2], David Parker[1] and Qi Hao[2,3]

***Abstract*— Shared autonomous mobility-on-demand systems hold great promise for improving the efficiency of urban transportation, but are challenging to implement due to the huge scheduling search space and highly dynamic nature of requests. This paper presents a novel optimal schedule pool (OSP) assignment approach to optimally dispatch high-capacity ride-sharing vehicles in real time, including: (1) an incremental search algorithm that can efficiently compute the exact lowest-cost schedule of a ride-sharing trip with a reduced search space; (2) an iterative online re-optimization strategy to dynamically alter the assignment policy for new incoming requests, in order to maximize the service rate. Experimental results based on New York City taxi data show that our proposed approach outperforms the state-of-the-art in terms of service rate and system scalability.**

I. Introduction

Ride-sharing is a promising solution for transportation issues such as traffic congestion and parking land use, which are brought about by the extensive usage of private cars. In the near future, large-scale shared autonomous mobility-on-demand (SAMoD) systems are expected to be deployed with the realization of self-driving vehicles [1], [2]. An SAMoD system consists of a fleet of shared self-driving vehicles and a centralized dispatch server receiving on-demand requests sent from passengers, as shown in Figure 1. The basic idea behind an SAMoD system is to assign suitable vehicles to each request and group multiple requests into ride-sharing trips if they are travelling in similar directions.

However, significant adoption of SAMoD systems requires the following technical challenges to be addressed:

1) *Large-scale.* A typical urban taxi system has over ten thousands vehicles. The state space of ride-sharing combinations and routes grows exponentially as the number of requests or the capacity of vehicles increases.
2) *Time-sensitive.* Passengers are sensitive to the time of service. Each request needs to be assigned in a few seconds and completed as soon as possible.
3) *Dynamic.* Requests are received continuously throughout the day, instead of being known in advance. An optimal assignment at a given time may not be the best when considering additional new requests.

*This work is partially supported by the Science and Technology Innovation Committee of Shenzhen City (No:JCYJ20200109141622964) and the Intel ICRI-IACV Research Fund (CG#52514373).

[1]School of Computer Science, University of Birmingham, Birmingham, UK {cxl776, d.a.parker}@cs.bham.ac.uk

[2]Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China haoq@sustech.edu.cn (Corresponding author: Qi Hao)

[3]Research Institute of Trustworthy Autonomous System, Southern University of Science and Technology, Shenzhen, China

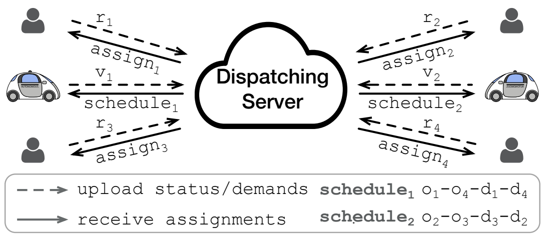

Fig. 1: Architecture of an SAMoD system with two vehicles and four requests, where vehicle 1 is assigned to serve requests 1&4 following schedule 1 (o_i: pick up request r_i, d_i: drop off r_i) and vehicle 2 is assigned to serve requests 2&3 following schedule 2.

Many approaches to controlling and analyzing SAMoD systems have been studied, such as multi-commodity flow models [3], queuing network models [4] and search-based models [5], [6]. Due to computational complexity, most existing work is restricted to small-scale and double-occupancy fleets for optimal assignments. Algorithms for optimal assignment usually formulate the problem as Mixed-Integer Linear Programming, which is impractical when thousands of vehicles are needed. Greedy methods have been used to accelerate the computation for large fleet ride-sharing, however, as the optimality of assignment is not guaranteed, the benefit of ride-sharing cannot be fully achieved.

This paper proposes an efficient online batch assignment scheme for dispatching high-capacity SAMoD systems in a practical timeframe. We use an incremental computation heuristic to reduce the search space of scheduling, and an iterative re-optimization procedure to dynamically and efficiently alter the assignment policy. The proposed approach is executable in real-time settings and can guarantee the optimality of the assignments at each dispatch epoch. To summarise our contributions, we:

1) Develop an incremental search algorithm to efficiently compute the optimal schedule of a ride-sharing trip, which reduces the global search to local search with heuristics while ensuring optimality (Sec. IV-A).
2) Combine the optimal schedule search algorithm with the feasible trip search algorithm of [5] to generate all possible ride-sharing trips for each vehicle, along with the optimal schedule for each trip (Sec. IV-B).
3) Develop an iterative re-optimization strategy to avoid myopic optimality, which takes into account both previous and new requests to optimize long-term system effectiveness (Sec. IV-C, IV-D).

4) Perform simulations with large-scale taxi data and evaluate our proposed approach, comparing to three representative algorithms (Sec. V).

II. Related Work

As requests appear throughout the day, online algorithms usually solve a static problem repeatedly in a rolling-horizon framework. Many approaches use greedy insertion, i.e., assigning one request at a time to the best-matched vehicle, to reduce the computational complexity, and most of them focus on the efficiency of dispatch. [7] introduces a grid-based index to accelerate the search of candidate vehicles, where a request only checks its nearby areas. [6] argues that the insertion of a new request into the route is the efficiency bottleneck, and proposes a dynamic programming algorithm to speed this up. These methods are efficient for large fleet and even high-capacity ride-sharing systems, but the quality of assignment cannot be guaranteed as the assignment of each request is greedy. [8] tries to use a replace procedure to achieve a better match quality by considering two requests at one time, but the improvement on service rate is very limited and the number of vehicles considered is reduced.

Batch assignment approaches collect the incoming requests over a period ΔT and compute the assignment of the collected requests simultaneously. Better performance is achieved than greedy insertion, but at the cost of much higher computational complexity. The simplest approach only computes all possible insertions of each single request to vehicles and then finds an optimal assignment of such insertions, as in e.g. [9]. That work also limits the maximum number of vehicles considered per request to handle thousands of vehicles, but its improvement over greedy insertion is relatively small. [10] computes an initial assignment of requests to vehicles, then performs random searches for more possible solutions to improve the quality of assignment. Inserting multiple requests into one vehicle's route is allowed in [10], but the instance scale solved is small. [5] generates all possible request insertions incrementally to break down the computational burden, and is able to dispatch thousands of high-capacity vehicles. However, a time limit is set in order to ensure efficiency and hence the state space is not guaranteed to be fully searched.

None of the above approaches tackle the optimal scheduling problem that finds the best order for picking and dropping multiple requests. Thus the optimality of assignment cannot be achieved as some requests may be mistakenly rejected. [3] develops a multi-commodity network flow model and formulates the assignment problem as a mixed integer linear program to get an optimal assignment. But the ride-sharing size is limited to two and the problem instances solved in the paper are relatively small. By contrast, we allow optimal dispatch of thousands of vehicles with a capacity of up to 10 in order to fully utilize the benefit of ride-sharing.

To further improve the performance of ride-sharing, some approaches work on sophisticated rebalancing method [11] or learning predictive value functions from historical data for better assignment [12]. They still suffer from the lack of an efficient and optimal generation of feasible solutions [12], and our approach, computing all optimal schedules, can help them have a better performance.

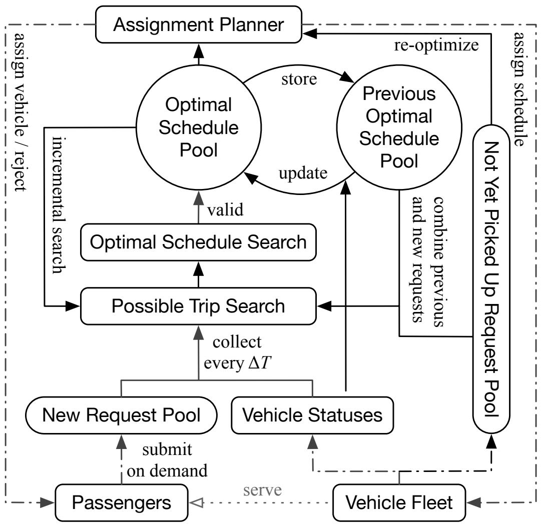

Fig. 2: Framework of the dispatch logic.

III. Problem Statement

A. Preliminaries

An instance of an SAMoD system consists of a fleet of m vehicles $V = \{v_1, \dots, v_m\}$ and a set of n new requests $R = \{r_1, \dots, r_n\}$ submitted. Each vehicle $v \in V$ is defined as a tuple $\langle q_v, \kappa_v, s_v \rangle$, where q_v is its current position, κ_v is the capacity of the vehicle and s_v is a planned schedule consisting of a sequence of pick-up and drop-off tasks. Each request $r \in R$ is defined as a tuple $\langle o_r, d_r, t_r \rangle$, where o_r is its origin, d_r is its destination, and t_r is the time the request is submitted. A waiting time ω_r (the difference between the actual pick-up time and the request sent time) and a total travel delay δ_r (the difference between the actual drop-off time and the travelling alone arrival time) are associated with each request.

A set of requests that can be merged through ride-sharing and served by a single vehicle is presented as a trip $\Gamma = \{r_1, \dots, r_{n_\Gamma}\}$. A trip might be served by more than one candidate vehicle and vice versa.

The order for a vehicle v to pick up and drop off requests in a trip Γ is defined as a schedule $s_{v,\Gamma} = \{q_v, \dots, o_1, \dots, d_1, \dots, o_2, \dots\}$, consisting of a sequence of visiting positions. There might be more than one feasible schedule for a specific combination of a vehicle v and a trip Γ; the set of all feasible schedules is denoted by $S_{v,\Gamma}$ and the minimum-cost one is denoted by $s^*_{v,\Gamma}$. A schedule must satisfy the following constraints:

- Each request must be served by exactly one vehicle.
- The pick-up point of each request must be visited before the drop-off point.
- The waiting time and travel delay of each request cannot exceed two thresholds, Ω and Λ, respectively.
- The number of passengers on board cannot exceed the capacity of the vehicle.

B. System Framework

Requests are collected periodically every ΔT (e.g., 30 seconds). Considering the current positions and schedules of the vehicle fleet, the dispatch server will search all possible ride-sharing trips for each vehicle. For each combination, the server verifies whether it is feasible, computes the optimal schedule and adds it to the optimal schedule pool. The optimal schedule pool is generated incrementally for increasing ride-sharing trip sizes. Then the server assigns each vehicle a schedule to serve requests. The assignment considers both new and previous received requests to maximize the system performance. A previous optimal schedule pool is maintained to save redundant computation. The details of the dispatch logic are shown in Figure 2.

IV. Optimal Online Dispatch Scheme

To tackle the optimal schedule search problem, we introduce an incremental search algorithm to reduce the scheduling space. It is further coupled with the possible trip search algorithm of [5] to efficiently generate the optimal schedule pool. A re-optimization strategy with speed-up heuristic is then presented to improve the long-term system performance.

A. Optimal Schedule

Computing the optimal schedule for a vehicle serving all requests in a trip is computationally expensive in general, as it is a generalization of the Travelling Salesman Problem with Precedence Constraints [13]. The following observation leads to the idea of incrementally searching all feasible schedules of a large trip.

Lemma 1: A schedule $s_{v,\Gamma}$ can be feasible only if any sub-schedule $s_{v,\Gamma}\backslash\{o_r, d_r\}$ (obtained by removing one request) of it is feasible, where $s_{v,\Gamma}\backslash\{o_r, d_r\} \in S_{v,\Gamma\backslash r}$. Therefore, a schedule $s_{v,\Gamma}$ only needs to be checked for satisfaction if, for any r in the schedule, the schedule set $S_{v,\Gamma\backslash r}$ is not empty.

Using Lemma 1, it is found that all $s_{v,\Gamma} \in S_{v,\Gamma}$ can be obtained through inserting a request r into some $s_{v,\Gamma\backslash r} \in S_{v,\Gamma\backslash r}$. We propose Algorithm 1 to efficiently compute the exact minimum-cost schedule of a high capacity ride-sharing trip Γ by only searching potentially feasible schedules instead of all schedule permutations. It generates the feasible schedule set $S_{v,\Gamma}$ incrementally by extending the initial schedule set and returns the optimal schedule $s^*_{v,\Gamma}$ or an empty schedule if there is no feasible schedule. S^a_v and S^b_v are define as two sets of all feasible schedules of vehicle v serving size $k-1$ and size k trips. In lines 6-7, it computes the set all the feasible schedules S^b_v by extending schedules in S^a_v. The function $InitScheduleSet(v)$ returns all the feasible schedules for the vehicle dropping passengers on board or $\{\{q_v\}\}$ if the vehicle is idle. The function $ScheduleInsersion(schedule, r)$ tries to insert o_r and d_r into all possible places to obtain new schedules and returns a set of all feasible new schedules or an empty set.

By pruning the infeasible subset of all possible schedules, finding the optimal schedule of a vehicle serving a trip is reduced from enumeration to a couple of basic schedule insertion processes. The tight constraints on maximum travel delay of SAMoD naturally narrows the solution space [2], which means that the size of any schedule set $S_{v,\Gamma}$ is normally small. Thus, the search space of possible schedules is significantly reduced.

Algorithm 1 Optimal Schedule Computing

```
Input   : a vehicle $v$ and a trip $\Gamma$
Output: the new optimal schedule $s^*_{v,\Gamma}$ for the vehicle $v$
 1: $S^a_v \leftarrow InitScheduleSet(v)$;
 2: $k \leftarrow 1$;
 3: while $S^a_v \neq \emptyset$ and $k <= n_\Gamma$ do
 4:     $S^b_v \leftarrow \emptyset$;
 5:     $r \leftarrow \Gamma.pop()$;
 6:     for each $schedule \in \mathcal{S}^a_v$ do
 7:         $S^b_v \leftarrow S^b_v \cup ScheduleInsersion(schedule, r)$;
 8:     $S^a_v \leftarrow S^b_v$;
 9:     $k \leftarrow k + 1$;
10: $s^*_{v,\Gamma} \leftarrow$ the minimum-cost schedule from $S^b_v$;
```

Algorithm 2 Optimal Schedule Pool Generating

```
Input   : a vehicle $v$ and a set of requests $R$
Output: the optimal schedule pool $\digamma^*_v$ for the vehicle $v$
 1: $\digamma^k_v \leftarrow \emptyset, \forall k \in \{0, 1, \ldots, \kappa_v\}$;
 2: $S_{v,\{\emptyset\}} \leftarrow InitScheduleSet(v)$;
 3: $\digamma^0_v \leftarrow \digamma^0_v \cup \{S_{v,\{\emptyset\}}\}$;
 4: for each $r \in R$ do
 5:     $S_{v,\{r\}} \leftarrow ScheduleSearch(S_{v,\{\emptyset\}}, r)$;
 6:     $\digamma^1_v \leftarrow \digamma^1_v \cup \{S_{v,\{r\}}\}$;
 7: $k \leftarrow 2$;
 8: while $\digamma^{k-1}_v \neq \emptyset$ and $k <= \kappa_v$ do
 9:     for all $S_{v,\Gamma_i}, S_{v,\Gamma_j} \in \digamma^{k-1}_v$ with $|\Gamma_i \cup \Gamma_j| = k$ do
10:         Denote $\Gamma_i \cup \Gamma_j = \Gamma^k = \Gamma_i \cup \{r_{new}\}$;
11:         $S_{v,\Gamma^k} \leftarrow ScheduleSearch(S_{v,\Gamma_i}, r_{new})$;
12:         $\digamma^k_v \leftarrow \digamma^k_v \cup \{S_{v,\Gamma^k}\}$;
13:     $k \leftarrow k + 1$;
14: $\digamma_v \leftarrow \digamma^1_v \cup \cdots \cup \digamma^{\kappa_v}_v$;
15: $\digamma^*_v \leftarrow ExtractOptimalSchedule(\digamma_v)$;
```

B. Optimal Schedule Pool

Borrowing the algorithmic idea of trip group feasibility from [5], we develop a procedure that can truly compute all feasible trips of a vehicle, along with the optimal schedule for each trip. It is illustrated in Algorithm 2, where $\digamma_v = \{S_{v,\Gamma_1}, S_{v,\Gamma_2}, S_{v,\Gamma_3}, \ldots\}$ is defined as the feasible schedule pool, containing all feasible schedule sets for vehicle v. Algorithm 2 incrementally computes $\digamma_v$ in increasing trip size and outputs the optimal schedule pool $\digamma^*_v = \{s^*_{v,\Gamma_1}, s^*_{v,\Gamma_2}, \ldots\}$, which contains all candidate optimal schedules of vehicle v serving all trips $\Gamma_i \subseteq R$. The function $ScheduleSearch(S_{v,\Gamma}, r)$ in lines 5 and 12, as in lines 6-7 of Algorithm 1, tries to compute schedules of trip $\Gamma \cup \{r\}$ and returns the set of feasible schedules $S_{v,\Gamma\cup\{r\}}$ if possible. Algorithm 2 can be parallelized among the vehicles.

Algorithm 2 further reduces the computation of $s^*_{v,\Gamma}$ to $|S_{v,\Gamma\backslash r}|$ calls to function $ScheduleInsersion(schedule, r)$. Many redundant computations are saved, as the feasible schedule set $S_{v,\Gamma}$ does not need to be computed completely from scratch on each request in trip Γ. By considering all feasible schedules for each trip, no trip would be mistakenly ignored and more feasible trips could be found than [5].

C. Iterative Updating

Although batch assignment approaches consider multiple requests simultaneously, there may still be cases where a later request, that can be served if the batch period is larger, will be mistakenly rejected. Therefore, re-optimization is introduced in order to escape from of a local minimum by removing the past assignment and computing a new one based on all known requests. However, this is too computationally expensive if we simply compute the optimal schedule pool for all received requests. So, we design an *updating heuristic* to reduce computation by generating the optimal schedule pool from previous schedules $\digamma^*_{v,prev}$.

The procedure is shown in Algorithm 3, where R_{prev} is defined to be the previous received requests and R_{new} is defined to be the new submitted ones. Function $UpdatePreviousSchedule(\digamma^*_{v,prev}, v)$ updates schedules in $\digamma^*_{v,prev}$ based on the current status of the vehicle, removes the infeasible ones and outputs the feasible ones in the same format as $\digamma_{v,prev}$. Function $Size1ScheduleSet(v, \mathcal{R}_{new})$ is equivalent to lines 4-6 in Algorithm 2 and computes the optimal schedule set $\digamma^1_v$ for R_{new}. Only computing combinations between the new requests and the previous ones, not among all known unpicked-up requests, makes the algorithm more efficient. Suppose we have n_{all} requests in the pool and n_{new} of them are newly submitted; the number of combinations for naive computation is $1/2 \cdot n_{all}(n_{all} - 1)$, while the number with the updating heuristic is $1/2 \cdot (2n_{all} - n_{new} - 1)n_{new}$. We define $z = n_{new}/n_{all}$, as normally $n_{new} \gg 1$, then:

$$\begin{aligned}\frac{1/2 \cdot (2n_{all} - n_{new} - 1)n_{new}}{1/2 \cdot n_{all}(n_{all} - 1)} &= \frac{2n_{new}}{n_{all}} - \frac{n_{new}(n_{new} - 1)}{n_{all}(n_{all} - 1)} \\ &= 2z - z\frac{n_{new} - 1}{n_{all} - 1} \\ &\approx 2z - z^2\end{aligned}$$

In our experiments (see the next section), the number of new requests is normally less than 500 while the number of all unpicked-up requests is up to 2600. There are significantly fewer combinations between 500 and 2600 requests than among 2600 requests. Based on the following observation, $\digamma^*_v$, as generated by Algorithm 3, still contains all possible optimal schedules.

Lemma 2: A schedule $s_{v,\Gamma}$ can be feasible at time t only if it is feasible at $t - \Delta T$. Thus, a feasible optimal schedule $s^*_{v,\Gamma}$ ($\forall\ \Gamma \subseteq R_{prev}$) at time t must is included in $\digamma^*_{v,prev}$.

With re-optimization, the SAMoD system can find a balance between the response time for requests and the optimality of the system. A small period yields a short response time, whereas considering all known requests makes the assignment policy optimal at any given time.

Algorithm 3 Schedule Pool Updating

```
Input  : a vehicle $v$ with its previous optimal schedule pool
         $\digamma^*_{v,prev}$ and a set of new requests $R_{new}$
Output: the new schedule pool $\digamma_v$
 1: $\digamma_{v,prev} \leftarrow UpdatePreviousSchedule(\digamma^*_{v,prev}, v)$;
 2: $\digamma^k_v \leftarrow \emptyset, \forall k \in \{0, 1, \dots, \kappa_v\}$;
 3: $\digamma^1_v \leftarrow Size1ScheduleSet(v, R_{new}) \cup \digamma^1_{v,prev}$;
 4: $k \leftarrow 2$;
 5: while $\digamma^{k-1}_v \neq \emptyset$ and $k <= \kappa_v$ do
 6:     $n_{new\_trip} = len(\digamma^{k-1}_v) - len(\digamma^{k-1}_{v,prev})$;
 7:     for $S_{v,\Gamma_i} \in \digamma^{k-1}_v[0 : n_{new\_trip}]$ do
 8:         for $S_{v,\Gamma_j} \in \digamma^{k-1}_v[1 :]$ do
 9:             Denote $\Gamma_i \cup \Gamma_j = \Gamma^k = \Gamma_i \cup \{r_{new}\}$;
10:             $\digamma^k_v \leftarrow \digamma^k_v \cup ScheduleSearch(S_{v,\Gamma_i}, r_{new})$;
11:     $\digamma^k_v \leftarrow \digamma^k_v \cup \digamma^k_{v,prev}$;
12:     $k \leftarrow k + 1$;
13: $\digamma_v \leftarrow \digamma^1_v \cup \dots \cup \digamma^{\kappa_v}_v$;
```

D. Constrained Optimization

Given a vehicle fleet V, a set of previous received yet not served requests R_{prev} and a batch of new incoming requests R_{new}, we define $R_{all} = R_{new} \cup R_{prev}$ and R_H representing the previous assigned requests. After an optimal schedule pool $\digamma^*_v$ has been computed for each vehicle, the goal is to assign each vehicle v a particular schedule s to serve as many requests as possible and minimize the overall travel cost raised by ride-sharing. This can be formulated as the solution of the following integer linear program (ILP):

$$\underset{x_{v,s},\epsilon_r}{\operatorname{argmin}} \quad \sum_{v\in V}\sum_{s\in \digamma^*_v} x_{v,s} \cdot c(s) + \sum_{r\in R_H} \epsilon_r \cdot p_{r_H} \tag{1}$$

$$s.t. \quad \sum_{s\in \digamma^*_v} x_{v,s} = 1, \quad \forall v \in V \tag{2}$$

$$\sum_{v\in V}\sum_{s\in \digamma^*_v} x_{v,s} \cdot \Theta_s(r) + \epsilon_r = 1, \quad \forall r \in R_{all} \tag{3}$$

where $c(s)$ is the cost of the planned schedule s, p_{r_H} is a very larger penalty for rejecting a request in R_H and $\Theta_s(r)$ is an indicator function, i.e., $\Theta_s(r) = 0$ if $r \notin s$ and $\Theta_s(r) = 1$ otherwise. Binary variable $x_{v,s} \in \{0, 1\}$ indicates whether a schedule is ignored or assigned to vehicle v and binary variable $\epsilon_r \in \{0, 1\}$ indicates whether a request is assigned or ignored. Constraint (2) enforces that each vehicle is assigned exactly one optimal schedule and constraint (3) enforces that each request can only be assigned to at most one vehicle. The cost of a schedule can be the mileage driven, the travel time or the revenue [6]. In this paper, the cost is defined as $c(s) = \sum_{r\in s}(\omega_r + \delta_r - p_r)$ to maximize the service rate as a priority, where p_r is a penalty smaller than R_H for rejecting a request.

E. Optimality

The generation of the optimal schedule pool exhaustively explores all combinations of currently received requests and vehicles. The saving of redundant computations makes it

TABLE I: Parameter settings (defaults in bold).

Parameters	Settings
Instance Scale $(\|R\|, \|V\|)$	**(400k, 2000)**, (500k, 2300), (600k, 2600), (700k, 2900), (800k, 3200)
Capacity κ	2, 4, 6, **8**, 10
Maximum Waiting Time Ω (s)	120, 180, 240, **300**, 360, 420
Batch Period ΔT (s)	2, 5, 10, **30**, 60, 120

possible to do this generation in real time. The computation of the optimal schedule for each trip ensures that no trip is mistakenly ignored. Then, solving the ILP presented above considers all possible assignment policies and returns the one that produces the minimal value of the objective function (1).

V. Experimental Study

In this section, our method is evaluated and compared to the leading online dispatch algorithm in [5] and two other representative algorithms [6], [9]. Since existing implementations are unavailable, we reimplement all algorithms and run them on the same machine to ensure a fair comparison.

A. Dataset

The experiments are conducted on the the largest public taxi dataset from New York City [14]. , which has been widely used in existing ride-sharing studies [2], [5], [6], [15], [11], [12]. We extracted requests data from three Wednesdays: 11th, 18th and 25th May in 2016. These have similar characteristics and we use them to synthesize five scenarios of varying size (400,000–800,000 requests) to test the scalability of algorithms.

B. Simulation Details

We simulate a ride-sharing environment following the settings in [5]. The initial locations of vehicles are uniformly distributed over the road network at the start of the experiment. The shortest paths among all nodes in the road network are pre-computed, using the daily mean travel time. Table I summarizes the major experimental parameters. As well as increasing the number of requests, we also increase the fleet size to have instances of realistic scale. The maximum travel delay is set to twice the maximum wait time. For various different capacities, we simulate the entire day. Then, for other parameters, we run the simulation for the hour with peak demand (19:00-20:00). This is the most challenging part and is enough to show the characteristics of the algorithms. The simulation implementation is single-threaded and the results are the averages of five experiment runs.

C. Algorithm Comparison

Our OSP algorithm is compared with the following representative algorithms:

- GI [6]: Greedy Insertion. This assigns requests sequentially to the best available vehicle in a first-in-first-out manner (i.e., an exhaustive version of [7]).
- SBA [9]: Single-Request Batch Assignment. This takes the new requests for a batch period and assigns them together in a one-to-one match manner, where at most one new request is assigned to a single vehicle.

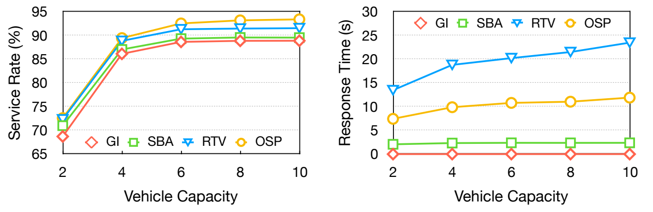

Fig. 3: Performance comparison during the whole day for varying capacity κ.

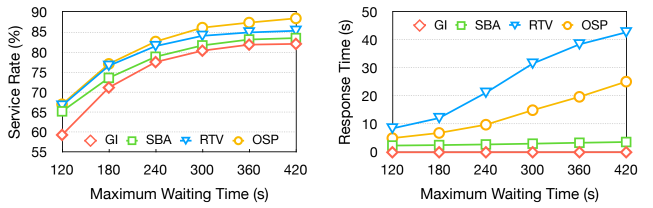

Fig. 4: Performance comparison during the peak hour for varying maximum waiting time Ω.

- RTV [5]: Request Trip Vehicle Graph. This is a more sophisticated batch assignment algorithm that copes with multiple request assignments during the same batch period. It uses ad hoc heuristics (e.g., only 30 candidate vehicles for each request, finding optimal schedules trips up to size 4 and a timeout of 0.2 s per vehicle to search feasible trips) to make it run in real-time.

All algorithms are equipped with the same simple rebalancing method from [5]. Our comparison focuses mainly on the service rate (percentage of requests served) and the response time (mean computational time of every batch period).

D. Results

Fig. 3 shows the results of varying vehicle capacity. As the capacity increases, all algorithms provide better service rates. Compared to double-occupancy, high-capacity ride-sharing significantly increases the service rate. All batch assignment methods outperform GI. Our OSP algorithm has the highest service rate and shows a 3.85% improvement over SBA at capacity 10. RTV is marginally worse than OSP when the capacity is less than 6, but it only achieves a 1.99% higher service rate than SBA at 10-capacity, which is only around half of the improvement brought by OSP. SBA outperforms GI more at double-occupancy than high capacity, because two seats would rarely let multiple request assignments happen during the same batch period. The gap between SBA and RTV/OSP at double-occupancy is mainly caused by re-optimization. The response times of GI and SBA are almost unchanged at varying capacity, while that of RTV and OSP linearly increase.

Fig. 4 plots the results of varying maximum waiting time. With a larger waiting time, the service rate of all algorithms increases. The reason is that a longer waiting time and travel delay allows a larger detour, and thus more requests are served. Batch assignment approaches achieve significantly higher service rates than GI when the maximum waiting time is low. But, when the tolerance to delay increases, the ability to leverage complex combinations of requests and

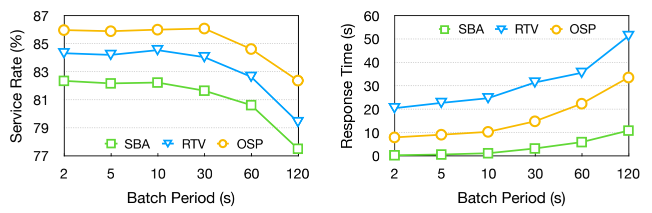

Fig. 5: Performance comparison during the peak hour for varying batch period ΔT.

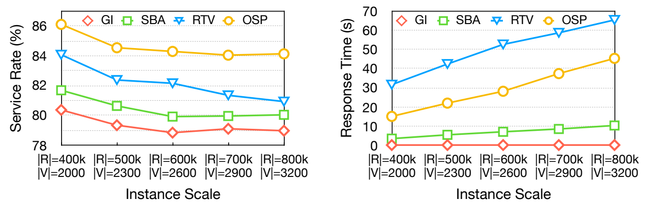

Fig. 6: Performance comparison during the peak hour for varying instance scale.

their optimal schedules is needed to maintain this advantage. The response time of GI and SBA is still stable. While the running time of OSP grows almost linearly, that of RTV increases exponentially. The rate of growth in RTV's running time decreases when $\Omega > 300$ s because the time limit slows down the increase in computational time.

Fig. 5 shows the results of varying batch period. With a larger period, more requests are processed at one epoch and thus the response time of batch approaches grows. Considering more requests in one period could theoretically have a better matching quality. However, a longer period yields a longer waiting and a lower detour tolerance, and results in a lower service rate. When the batch period is longer than 60 s, the service rates decrease significantly.

Fig. 6 plots the results of varying instance scale. All algorithms scale well to large problem instances with a linear increase in response time. However, the service rate of RTV has a considerable decrease compared to others. The difference between OSP and RTV is expanded from 2.04% to 3.21$\%$. This indicates that, although the ad hoc heuristics used by RTV ensure a good scalability on response time, as optimality cannot be guaranteed, they also bring a worse matching performance at large instance scale.

Fig. 7 shows the total number of feasible trips found and the number of matched new requests by different algorithms from 19:30 to 20:00, where the average number of incoming new requests is 438 per 30 s. In each dispatch period, all algorithms are fed with the exact same requests and vehicles to remove other distractions (e.g. the position distribution of requests). RTV-FULL, without time limit and allowed to keep every feasible vehicle for each request, is introduced to have a comprehensive comparison between OSP and RTV. It can be seen that OSP has the best performance at each dispatch epoch, as it finds every feasible trip with the optimal schedule and makes the best match based on them. The difference between OSP and RTV-FULL is caused by the fact that RTV-FULL only does exhaustive search for trips up to size 4. Despite this, the response time of RTV-FULL is 763.98 s, while that of RTV and OSP are 65.48 s and

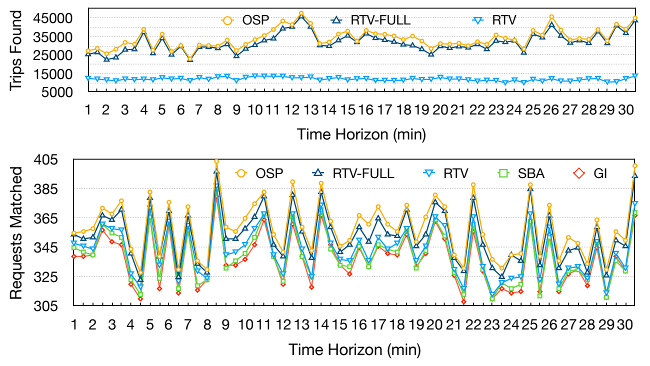

Fig. 7: Number of feasible trips found for all vehicles (upper plot) and number of matched new submitted requests (lower plot) at different dispatch epochs ($|R| = 800k, |V| = 3200, \kappa = 8, \Omega = 300$ s$, \Delta T = 30$ s).

TABLE II: Number of schedules searched by algorithms

Trip Size	3	4	5	6	7
Exhaustive Search	54	1150	6983	53,110	272,810
OSP (ours)	11	50	159	433	595

45.38 s. If we allow RTV-FULL to do exhaustive search for size 5 trips, the response time increases to 3,734.77 s.

We further investigate the number of possible schedules considered by OSP to see how much of the search space is pruned compared to exhaustive search. Table II shows the counts from trips of size 3 to 7. Exhaustive search is unable to complete the search when the trip size is larger than 7, while OSP only considers 1,920 schedules for size 10 trips.

E. Summary

Batch assignment provides better performance than GI for assigning requests together. Additionally, our OSP's optimal scheduling can achieve twice the improvement of RTV over SBA. The improvement of OSP relative to RTV also rises with the complexity of the instance. The incremental schedule search method and the iterative re-optimization strategy together reduce the scheduling space to less than one percent of that of exhaustive search, when the ride-sharing size is larger than six. The computation of the schedule pool is naturally parallelized among the vehicles, thus OSP is able to be deployed in real-time with even larger instance scale.

VI. Conclusion

In this paper, we have proposed an optimal online scheduling algorithm (OSP) for high capacity SAMoD systems incorporating an incremental schedule search algorithm and an iterative re-optimization strategy. Numerical experiments on real large-scale datasets show the proposed method improves the state-of-the-art in terms of service rate (up to 3.21$\%$ at peak hour) and system scalability. Typically, a 1$\%$ improvement is considered significant on real taxi systems [12], [16]. Our work aims to study reactive optimal batch assignment for SAMoD systems running in real time, and tries to reach the upper-bound of performance for reactive dispatch. Future work will investigate the combination of complete feasible trip search and predictive control.

References

[1] M. Pavone, “Autonomous mobility-on-demand systems for future urban mobility,” in *Autonomes Fahren*, pp. 399–416, Springer, 2015.

[2] M. Cáp and J. Alonso-Mora, “Multi-objective analysis of ridesharing in automated mobility-on-demand,” in *Robotics: Science and Systems*, 2018.

[3] M. Tsao, D. Milojevic, C. Ruch, M. Salazar, E. Frazzoli, and M. Pavone, “Model predictive control of ride-sharing autonomous mobility on demand systems,” in *Proc. IEEE Conf. on Robotics and Automation*, 2019.

[4] R. Zhang and M. Pavone, “Control of robotic mobility-on-demand systems: a queueing-theoretical perspective,” *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 186–203, 2016.

[5] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus, “On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment,” *Proceedings of the National Academy of Sciences*, vol. 114, no. 3, pp. 462–467, 2017.

[6] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu, “A unified approach to route planning for shared mobility,” *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1633–1646, 2018.

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

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

[9] A. Simonetto, J. Monteil, and C. Gambella, “Real-time city-scale ridesharing via linear assignment problems,” *Transportation Research Part C: Emerging Technologies*, vol. 101, pp. 208–232, 2019.

[10] J. Jung, R. Jayakrishnan, and J. Y. Park, “Dynamic shared-taxi dispatch algorithm with hybrid-simulated annealing,” *Computer-Aided Civil and Infrastructure Engineering*, vol. 31, no. 4, pp. 275–291, 2016.

[11] A. Wallar, M. Van Der Zee, J. Alonso-Mora, and D. Rus, “Vehicle rebalancing for mobility-on-demand systems with ride-sharing,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4539–4546, IEEE, 2018.

[12] S. Shah, M. Lowalekar, and P. Varakantham, “Neural approximate dynamic programming for on-demand ride-pooling,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 507–515, 2020.

[13] J. J. Pan, G. Li, and J. Hu, “Ridesharing: simulator, benchmark, and evaluation,” *Proceedings of the VLDB Endowment*, vol. 12, no. 10, pp. 1085–1098, 2019.

[14] N. Taxi and L. Commission, “Tlc trip record data.” `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`, June 2019. Accessed: 2020-01-12.

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

[16] Z. Xu, Z. Li, Q. Guan, D. Zhang, Q. Li, J. Nan, C. Liu, W. Bian, and J. Ye, “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, 2018.