Dynamic Ridehailing with Electric Vehicles

Nicholas D. Kullman[1], Martin Cousineau[2], Justin C. Goodson[3], and Jorge E. Mendoza[2]

[1] *Université de Tours, LIFAT EA 6300, CNRS, ROOT ERL CNRS 7002, Tours, France*
[2] *HEC Montréal and CIRRELT, Montréal, Canada*
[3] *Richard A. Chaifetz School of Business, Saint Louis University, Saint Louis, MO*

Abstract

We consider the problem of an operator controlling a fleet of electric vehicles for use in a ridehailing service. The operator, seeking to maximize profit, must assign vehicles to requests as they arise as well as recharge and reposition vehicles in anticipation of future requests. To solve this problem, we employ deep reinforcement learning, developing policies whose decision making uses Q-value approximations learned by deep neural networks. We compare these policies against a reoptimization-based policy and against dual bounds on the value of an optimal policy, including the value of an optimal policy with perfect information which we establish using a Benders-based decomposition. We assess performance on instances derived from real data for the island of Manhattan in New York City. We find that, across instances of varying size, our best policy trained with deep reinforcement learning outperforms the reoptimization approach. We also provide evidence that this policy may be effectively scaled and deployed on larger instances without retraining.

1 Introduction

Governmental regulations as well as a growing population of environmentally conscious consumers have led to increased pressure for firms to act sustainably. This pressure is particularly high in the logistics domain, which accounts for about one third of emissions in the United States (Office of Transportation and Air Quality 2019). Ridehailing services offer a means to more sustainable transportation, promising to reduce the need for vehicle ownership, offering higher vehicle utilization (Lyft 2018), allowing transit authorities to streamline services (Bahrami et al. 2017), and stimulating the adoption of new vehicle technologies (Jones and Leibowicz 2019). In recent years, ridehailing services have seen rapid and widespread adoption, with the number of daily ridehailing trips more than quadrupling in New York City from November 2015 to November 2019 (New York City Taxi & Limousine Commission 2018).

Simultaneously, electric vehicles (EVs) are beginning to replace internal-combustion engine (ICE) vehicles, commanding increasingly more market share (Edison Electric Institute 2019). Coupling the pressures to act sustainably with EVs' promise of lower operating costs, ridehail companies are likely to be among the largest and earliest adopters of EV technology. Indeed most major ridehail companies have made public commitments to significant EV adoption (Slowik et al. 2019). However, EVs pose technological challenges to which their ICE counterparts are immune, such as long recharging times and limited recharging infrastructure (Pelletier et al. 2016).

In this work, we consider these challenges as posed to an operator of a ridehail company whose fleet consists of EVs. We further assume that the EVs in the fleet are centrally controlled and coordinated.

While fleet control is somewhat centralized today, it is likely to become increasingly centralized as ridehail companies adopt autonomous vehicles (AVs). As with EVs, ridehail companies are likely to be among the earliest and largest adopters of AVs, as they stand to offer many benefits including reduced operating costs and greater efficiency and predictability (Fagnant and Kockelman 2015). For brevity, we refer to this problem as the Electric Ridehail Problem with Centralized control, or E-RPC.

Our contributions in this work are as follows: 1) We offer the first application of deep reinforcement learning (RL) to the E-RPC, developing policies that respond in real time to serve customer requests and anticipate uncertain future demand under the additional constraints of fleet electrification. The policies are *model-free*, meaning they learn to anticipate this demand without any prior knowledge of its shape. We compare these deep RL-based policies to a reoptimization-based policy from the ridehailing literature. 2) We evaluate these policies on instances constructed with real data reflecting ridehailing operations from New York City in 2018. 3) We establish a dual bound for the dynamic policies using a perfect information relaxation that we solve using a Benders-like decomposition. We compare this complex dual bound against a simpler dual bound and provide an analysis of when the additional complexity of the perfect information bound is valuable. 4) We show that our best deep RL-based solution significantly outperforms the benchmark reoptimization policy. 5) We further demonstrate that the best-performing deep RL-based policy can be scaled to larger problem instances without additional training. This is encouraging for operators of ridehail companies, as it suggests robustness to changes in the scale of operations: in the event of atypical demand or a change in the number of vehicles (e.g., due to fleet maintenance), the policy should still provide reliable service.

We begin by reviewing related literature in §2, then provide a formal problem and model definition in §3. We describe our solution methods in §4, the bounds established to gauge the effectiveness of these methods in §5, and demonstrate their application in computational experiments in §6. We offer brief concluding remarks in §7.

2 Related Literature

Ridehail problems (RPs), those addressing the operation of a ridehailing company, fall under the broader category of dynamic vehicle routing problems (VRPs). Within dynamic VRPs, they may be classified as a special case of the dynamic VRP with pickups and deliveries or, more precisely, a special case of the dynamic dial-a-ride problem. Recent technological advances in mobile communications have driven new opportunities in ridehailing and other *mobility-on-demand* (MoD) services, reinvigorating research in this area. As a result, there now exists a substantial body of literature specifically pertaining to MoD applications within the dial-a-ride domain. This literature is the focus of our review. For a broader survey of the dynamic dial-a-ride literature, we refer the reader to Ho et al. (2018), and, similarly, for the dynamic VRP literature, to Psaraftis et al. (2016).

Often studies of RPs focus their investigation on the assignment problem, wherein the operator must choose how to assign fleet vehicles to new requests. For example, Lee et al. (2004) propose the use of real-time traffic data to assign the vehicle which can reach the request fastest. A study by Bischoff and Maciejewski (2016) uses a variant of this assignment heuristic that accounts for whether demand exceeds supply (or vice versa), as well as vehicles' locations within the service region. Seow et al. (2009) propose a decentralized heuristic that gathers multiple requests and allows vehicles to negotiate with one another to determine which vehicles serve the new requests. They show that it outperforms heuristics like those in Lee et al. (2004) and Bischoff and Maciejewski (2016). Hyland and Mahmassani (2018) introduce a

suite of optimization-based assignment strategies and show that they outperform heuristics that do not employ optimization. Following suit, Bertsimas et al. (2019) describe ways to reduce the complexity of optimization-based approaches for problems with large fleets and many customers. They demonstrate their proposed methods on realistically-sized problem instances that reflect ridehailing operations in New York City.

The aforementioned studies largely ignore the task of repositioning idle vehicles in anticipation of future demand. This is in contrast to studies such as Miao et al. (2016), Zhang and Pavone (2016), and Braverman et al. (2019) who address their RP from the opposing perspective of the fleet repositioning problem. Miao et al. (2016) do so using learned demand data with a receding horizon control approach. Braverman et al. (2019) use fluid-based optimization methods for the repositioning of idle vehicles to maximize the expected value of requests served. They establish the optimal static policy and prove that it serves as a dual bound for all (static or dynamic) policies under certain conditions. Other studies consider strategies for both the assignment and repositioning tasks. Fagnant and Kockelman (2014) propose rule-based heuristic strategies while most others (such as Alonso-Mora et al. 2017, Dandl et al. 2019, and Hörl et al. 2019) employ various optimization-based methods. Because it is scalable and well-suited for adaptation to our problem model, we implement the approach from Alonso-Mora et al. (2017) here as a benchmark against which to compare our proposed solution methods.

A recent competing method to address ridehail problems is reinforcement learning (RL), predominantly deep reinforcement learning. As it is better-positioned to address problems with larger fleet sizes, many studies employing RL take a multi-agent RL (MARL) approach (Oda and Tachibana 2018, Oda and Joe-Wong 2018, Holler et al. 2019, Li et al. 2019, Singh et al. 2019). In work sharing many similarities to ours, Holler et al. (2019) use deep MARL with an attention mechanism to address both the assignment and repositioning tasks. They compare performance under two different MARL approaches: one in which a system-level agent coordinates vehicle actions to maximize total reward; and one in which vehicle-level agents act individually, each maximizing its own reward. Oda and Joe-Wong (2018) use a deep MARL approach to tackle fleet repositioning, but rely on a myopic heuristic to perform assignment. Oda and Tachibana (2018) do so as well, but employ special network architectures with soft-Q learning which they argue better accommodates the inherent complexities in road networks and traffic conditions. Li et al. (2019) employ a multi-agent RL framework, comparing two approaches that vary in what individual vehicles know regarding the remainder of the fleet; however, in contrast to most MARL applications to RPs, they address only the assignment problem, assuming that the vehicles are de-centralized and therefore not repositioned by the operator. Similarly assuming de-centralized vehicles and addressing only the assignment problem, Xu et al. (2018) combine deep RL with the optimization-based methods used in, e.g., Alonso-Mora et al. (2017), Zhang et al. (2017), and Hyland and Mahmassani (2018). In their work, Xu et al. use deep RL to learn the objective coefficients in a math program that is used in an optimization framework to assign vehicles to requests.

Missing from all aforementioned studies is fleet electrification — none take into account the technical challenges associated with electric vehicles, such as long recharging times or limited recharging infrastructure. The number of studies that consider these constraints is limited but growing. Most (e.g., Jung et al. 2012; Chen et al. 2016; Kang et al. 2017; Iacobucci et al. 2019; La Rocca and Cordeau 2019) use the heuristic and optimization methods previously mentioned. We focus here on three works closely related to ours that employ learning-based approaches. First, Al-Kanj et al. (2018) use approximate dynamic programming to learn a hierarchical interpretation of a value function that is used to determine whether or not to recharge vehicles at their current location, in addition to repositioning decisions to neighboring grid cells and the assignment of vehicles to nearby requests. The study examines the possi-

bility for riders or the operator to reject a trip assignment at various costs; it also studies the problem of fleet sizing. Second, Pettit et al. (2019) use deep RL to learn a policy determining when an EV should recharge its battery and when it should serve a new request. The study includes time-dependent energy costs. However, it only considers a single vehicle, and it ignores the task of repositioning the vehicle in anticipation of future requests. Finally, Shi et al. (2019) apply deep reinforcement learning to the RP with a community-owned fleet, in which they seek to minimize customer waiting times and electricity and operating costs. Similar to many of the RL studies cited previously, the study employs a multi-agent RL framework that produces actions for each vehicle individually which are then used in a centralized decision-making mechanism to produce a joint action for the fleet. The joint actions assign vehicles to requests and specify when vehicles should recharge; however the study also ignores repositioning actions.

Thus, ours marks the first application of deep reinforcement learning to the E-RPC in consideration of EV constraints alongside both fleet repositioning and assignment tasks. We also argue that we consider a more realistic problem environment than in most RL-based approaches to ridehail problems. Indeed, the vast majority of studies (e.g., Al-Kanj et al. 2018; Oda and Tachibana 2018; Holler et al. 2019; Shi et al. 2019; Singh et al. 2019) rely on a coarse grid-like discretization to describe the underlying service region. This serves as the basis for their repositioning actions, which are often restricted to only the adjacent grid cells. This is in contrast to the current work, in which we allow repositioning to non-adjacent sites that correspond to real charging station locations. Time discretization can also be overly coarse. Many studies choose one minute between decision epochs, as in, for example, Oda and Joe-Wong (2018); but some choose far longer between decision epochs, such as six minutes in Shi et al. (2019) and 15 minutes in Al-Kanj et al. (2018). Given that current ridehailing applications provide fast, sub-minute responses, such an arrangement would likely lead to customer dissatisfaction. We are free of such a discretization here, allowing agents to respond to customer requests immediately.

3 Problem Definition and Model

We consider a central operator controlling a homogeneous fleet of electric vehicles $\mathcal{V} = \{1, 2, \ldots, V\}$ that serve trip requests arising randomly across a given geographical region and over an operating horizon T. Across the region are uncapacitated charging stations (CSs, or simply *stations*) $\mathcal{C} = \{1, 2, \ldots, C\}$ at which vehicles may recharge or idle when not in service. The operator assigns vehicles to requests as they arise. Additionally, the operator manages recharging and repositioning operations in anticipation of future requests. The objective is to maximize expected profit across the operating horizon.

Several assumptions and conditions connect the problem to real-world operations. When a new request arises, the operator responds immediately, either rejecting the request or assigning it to a vehicle. A vehicle is eligible to serve a new request if it can pickup within w^* time units, the amount of time customers are willing to wait. Each vehicle maintains at most one pending trip request. Thus, a vehicle either serves a newly assigned request immediately or does so after completing the service of a request to which it is already assigned ("work-in-process"). Assignments of requests to vehicles cannot be canceled, rescheduled, or reassigned. Each vehicle has a maximum battery capacity Q and known charging rate, energy consumption rate, and speed. The operator ensures assignments and repositioning movements are energy feasible.

We formulate the problem as a Markov decision process (MDP) whose components are defined as follows.

States.

A *decision epoch* $k \in \{0, 1, \ldots, K\}$ begins with a new trip request or when a vehicle finishes its work-in-process, whichever occurs first. The system state is captured by the tuple

$$s = (s_t, s_r, s_{\mathcal{V}}). \tag{1}$$

The current time $s_t = (t, d)$ is the time of day in seconds t and the day of week d. If the epoch was triggered by a new request, then $s_r = ((o_x, o_y), (d_x, d_y))$ consists of the Cartesian coordinates for the request's origin and destination, otherwise $s_r = \emptyset$. The vector $s_{\mathcal{V}} = (s_v)_{v \in \mathcal{V}}$ describes the state of each vehicle in the fleet. For a vehicle v, $s_v = (x, y, q, j_m^{(1)}, j_o^{(1)}, j_d^{(1)}, j_m^{(2)}, j_o^{(2)}, j_d^{(2)}, j_m^{(3)}, j_o^{(3)}, j_d^{(3)})$, consisting of the Cartesian coordinates of the vehicle's current position x, y; its charge $q \in [0, Q]$; and a description of its current activity (or *job*) $j^{(1)}$, as well as potential subsequent jobs $j^{(2)}$ and $j^{(3)}$, which may or may not exist, as determined by the operator. The description of a job $j^{(i)}$ consists of its type $j_m^{(i)} \in \{\text{idle}, \text{charge}, \text{reposition}, \text{preprocess}, \text{serve}, \emptyset\}$ (equivalently, $\{0, 1, 2, 3, 4, \emptyset\}$; "preprocess" refers to when a vehicle is en route to pickup a customer), as well as the coordinates of the job's origin $j_o^{(i)} = (j_{o,x}^{(i)}, j_{o,y}^{(i)})$ and destination $j_d^{(i)} = (j_{d,x}^{(i)}, j_{d,y}^{(i)})$. We limit the number of tracked jobs in the state to three, because this is the maximum number of scheduled jobs a vehicle can have given the constraint that a vehicle may have at most one pending trip request: if a vehicle is currently serving a request and has a pending request, then it will have jobs of type $j_m^{(1)} = 4$, $j_m^{(2)} = 3$, and $j_m^{(3)} = 4$. Note that we do not allow vehicles currently preprocessing for one request to be assigned a second (this would indeed require tracking a fourth job). With a sufficiently strict customer service requirement (i.e., the time between receipt of a customer's request and a vehicle's arrival to the customer), the number of opportunities to meet that requirement after undergoing the necessary (preprocessing, serving, preprocessing) sequence is small enough as to warrant the situation's exclusion. If a vehicle has $n < 3$ scheduled jobs, then we set $j_m^{(i)} = j_o^{(i)} = j_d^{(i)} = \emptyset$ for $n < i \leq 3$. We note that a null job $j_m^{(i)} = \emptyset$ is distinct from an idle job $j_m^{(i)} = 0$. A null job is the absence of an assignment, whereas an idle job reflects an assignment to reside at a particular location.

An *episode* begins with the system initialized in state s_0 in epoch 0 at time 0 on some day d with no new request ($s_r = \emptyset$) and all vehicles idling with some charge at a station ($v \in \mathcal{C}$, $q \in [0, Q]$, $j_m^{(1)} = \text{idle}$, $j_m^{(2)} = j_m^{(3)} = \emptyset$ for all $v \in \mathcal{V}$). The episode terminates at some epoch K in state s_K when the time horizon T has been reached.

Actions.

When the process occupies state s, the set of actions $\mathcal{A}(s)$ defines feasible assignments of vehicles to a new request as well as potential repositioning and recharging movements. An action $\mathbf{a} = (a_r, a_1, \ldots, a_v, \ldots, a_V)$ is a vector comprised of components $a_r \in \mathcal{V} \cup \{\emptyset\}$ denoting which vehicle serves new request s_r as well as a repositioning/recharging assignment a_v for each vehicle v indicating the station to which it should be repositioned and begin recharging $a_v \in \mathcal{C} \cup \{\emptyset\}$. $a_r = \emptyset$ denotes rejection of the new request (if it exists) and $a_v = \emptyset$ denotes the "no operation" (NOOP) action for vehicle v, meaning it proceeds to carry out its currently assigned jobs. We refer to the vehicle-specific repositioning/recharging/NOOP actions a_v as RNR actions. Jobs of type *idle* (0), *charge* (1), *reposition* (2), and $\emptyset$ are preemptable, meaning they can be interrupted by a new assignment from the operator, whereas jobs of type *preprocess* (3) and *serve* (4) are non-preemptable.

The following conditions characterize $\mathcal{A}(s)$. If there is no new request ($s_r = \emptyset$), then $a_r = \emptyset$. If there is a new request, then a vehicle is eligible for assignment if it can be dispatched immediately

Table 1: Eligible actions for an EV v.

Action	Eligibility conditions
Serve request	Request exists, no pending service ($j_m^{(2)} < 3$), energy feasible, time feasible
Reposition to a_v	No active or pending service ($j_m^{(1)} < 3$), energy feasible
NOOP	Always, unless just finished serving and no pending service ($j_m^{(1)} = \emptyset$)

($j_m^{(1)} \in \{0, 1, 2, \emptyset\}$) or if it is currently servicing a customer ($j_m^{(1)} = 4$) and can then be dispatched ($j_m^{(2)} \notin \{3, 4\}$). Further, a vehicle-request assignment must be energy feasible and must arrive to the customer within w^* time units, the maximum amount of time a customer is willing to wait. Let $f_q(s_v, s_r)$ be the charge with which vehicle v will reach a station after serving request s_r. $f_q(s_v, s_r)$ is equal to the current charge of vehicle v, minus the charge required for the vehicle to travel to the origin of the new request (minus the charge needed to first drop off its current customer, if $j_m^{(1)} = 4$), to the destination of the new request, and then to the nearest $c \in \mathcal{C}$ from the destination. Let $f_t(s_v, s_r)$ be the duration of time required for vehicle v to arrive at the customer. $f_t(s_v, s_r)$ is equal to the time for vehicle v to travel to the origin of the new request from its current location (plus the time to first drop off its current customer, if $j_m^{(1)} = 4$). An assignment of vehicle v to request s_r is feasible if $f_q(s_v, s_r) \geq 0$ and $f_t(s_v, s_r) \leq w^*$. Repositioning and recharging movements must also be energy feasible and may not preempt existing service assignments. If a vehicle is currently serving or preparing to serve a request ($j_m^{(1)} \in \{3, 4\}$), or if the vehicle has been selected to serve the new request ($a_r = v$), then we only allow the NOOP action $a_v = \emptyset$. Otherwise, we allow repositioning to any station $c \in \mathcal{C}$ that can be reached given the vehicle's current charge level. Lastly, if a vehicle completes a trip request, triggering a new epoch, and that vehicle has not been assigned a subsequent trip request ($j_m^{(1)} = \emptyset$), then it must receive a repositioning action ($a_v \neq \emptyset$). This condition forces vehicles to idle only at stations. While this may seem restrictive, we note (i) that the instances in our computational experiments (§6) contain a high density of stations, (ii) that cities would likely prefer that idling ridehail vehicles do so only at dedicated locations, and (iii) that non-CS idling locations can easily be included in our model as CSs at which energy is replenished at a sufficiently small rate. We summarize the eligible actions for an EV in Table 1.

Reward Function.

When the process occupies state s and action $\mathbf{a}$ is selected, we earn the reward $C(s, \mathbf{a})$. This reward is equal to the revenue generated by serving a new request (if $a_r \neq \emptyset$), minus any traveling costs incurred during the subsequent epoch. Specifically, the revenue earned by serving new request s_r consists of a base fare C_B and a fare proportional to the distance of the request $C_D \cdot d(s_r)$, where C_D is the per-distance revenue and $d(s_r)$ is the distance from the request's origin to its destination. The traveling costs incurred are $C_T \cdot \hat{\delta}(s, \mathbf{a})$, where C_T is a per-distance travel cost incurred by the operator and $\hat{\delta}(s, \mathbf{a})$ is the total distance traveled by the fleet in the subsequent epoch, including all repositioning and service-related travel. We note that this value is stochastic, since the time of the next epoch is unknown and equal to either the time of the arrival of the next request, or the next time at which a vehicle completes work-in-process, whichever comes first.

This yields the following reward function:

$$C(s, \mathbf{a}) = \begin{cases} -C_T \cdot \hat{\delta}(s, \mathbf{a}) & a_r = \emptyset \\ C_B + C_D d(s_r) - C_T \cdot \hat{\delta}(s, \mathbf{a}) & \text{otherwise,} \end{cases} \tag{2}$$

Transition Function.

The transition from one state to the next is a function of the selected action and the event triggering the next epoch. The subsequent state s' is constructed by updating the current state s to reflect changes to vehicles' job descriptions based on selected action $\mathbf{a}$. Then, at time s'_t, we observe either a new request s'_r or a vehicle completing its work-in-process. At that time, we update the vehicle states $s'_{\mathcal{V}}$ to reflect new positions and charges. We also update vehicles' job descriptions: if a vehicle has completed n jobs between time s_t and s'_t, we left-shift vehicles' job descriptions by n ($j^{(i)} \leftarrow j^{(i+n)}$ for $1 \leq i \leq 3-n$) and backfill the vacated entries with null jobs ($j_m^{(i)} = j_o^{(i)} = j_d^{(i)} = \emptyset$ for $3-n < i \leq 3$). See Appendix A for a more detailed description.

Objective Function.

Define a policy π to be a sequence of decision rules $(X_0^\pi, X_1^\pi, \ldots, X_K^\pi)$, where X_k^π is a function mapping state s in epoch k to an action $\mathbf{a}$ in $\mathcal{A}(s)$. We seek to provide the fleet operator with a policy $\pi^\star$ that maximizes the expected total rewards earned during the time horizon, conditional on the initial state:

$$\tau(\pi^\star) = \max_{\pi \in \Pi} \mathbb{E}\left[\sum_{k=0}^{K} C(s_k, X_k^\pi(s_k)) \middle| s_0\right], \tag{3}$$

where Π is the set of all policies. As is common in reinforcement learning, we will often refer to the user or implementer of a policy as an *agent*.

4 Solution Methods

We consider four policies for solving the E-RPC (to which we will often refer synonymously as "agents"). These include two policies developed using deep reinforcement learning, which we describe in §4.1, and two benchmark policies, which we describe in §4.2.

4.1 Deep Reinforcement Learning Policies

Our two primary policies of interest leverage deep reinforcement learning (RL). The first of these uses deep RL to determine which vehicle to assign to new trip requests and relies on optimization-based logic for RNR decisions. We refer to this agent as *deep ART* (for "Assigns Requests To vehicles") or just *Dart*. Our second agent we dub *deep RAFTR* ("Request Assignments and FleeT Repositioning") or *Drafter*. Drafter uses deep RL both to assign vehicles to requests and to reposition vehicles in the fleet.

Before describing these policies in §4.1.3 and §4.1.4, we provide an overview of the basic method of deep reinforcement learning in §4.1.1 and a discussion of extensions to this basic method in §4.1.2.

4.1.1 Overview.

As defined in Sutton and Barto (2018), reinforcement learning refers to the process through which an *agent*, sequentially interacting with some environment, learns what to do so as to maximize a numerical reward signal from the environment; that is, learns a policy satisfying equation (3). In practice, the agent often achieves this by learning the value of choosing an action a from some state s, known as the state-action pair's *Q-value* ($Q(s, a)$), equal to the immediate reward plus the expected sum of future rewards earned by taking action a from state s. We can express this relationship recursively following

from the Bellman equation:

$$Q(s,a) = C(s,a) + \gamma \mathbb{E}\left[\max_{a' \in \mathcal{A}(s')} Q(s',a') \middle| s, a \right], \tag{4}$$

where s' is the subsequent state reached by taking action a from state s and $\gamma \in [0,1)$ is a discount factor applied to the value of future rewards. With knowledge of Q-values for all state-action pairs it may encounter, the agent's policy is then to choose the action with the largest Q-value. However, as the number of unique state-action pairs is too large to learn and store a value for each, a functional approximation of these Q-values is learned. When deep artificial neural networks are used for this approximation, the method is called *deep reinforcement learning* or, more specifically, *deep Q-learning* (we use the terms interchangeably here). The neural network used in this process is referred to as the *deep Q-network* (DQN).

Beginning with arbitrary Q-value approximations, the agent's DQN improves through its interactions with the environment, remembering observed rewards and state transitions, and drawing on these memories to update the set of weights that define the DQN, conventionally denoted by θ. Specifically, with each *step* (completion of an epoch) in the environment, the agent stores a memory which is a tuple of the form (s, a, r, s'), consisting of a state s, the action a taken from s, the reward earned $r = C(s,a)$, and the subsequent state reached s'. Once a sufficient number of memories (defined by the hyperparameter M_{start}) have been accumulated, the agent begins undergoing *experience replay* every M_{freq} steps (Lin 1992). In experience replay, the agent draws a random sample (of size M_{batch}) from its accumulated memories and uses them to update its DQN via stochastic gradient descent (SGD) on its weights θ using a loss function based on the difference

$$r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s',a') - Q(s,a) \tag{5}$$

(or simply $r - Q(s,a)$ if s' is terminal), where the Q-values in equation (5) are estimated using the agent's DQN. The specific loss function and SGD optimizer used to update the weights may vary — we provide the specific implementations chosen for this work in §B.

We utilize the learned Q-values via an *ϵ-greedy* policy, wherein the agent chooses actions randomly with some probability ϵ and chooses the action with the highest predicted Q-value with probability $1-\epsilon$. The value of ϵ is decayed from some initially large value ϵ_i at the beginning of training to some small final value ϵ_f over the course of some number ϵ_N of training steps. This encourages exploration when Q-values are unknown and exploitation as its predictions improve.

4.1.2 Implemented extensions to deep RL.

We adopt several well established extensions that improve the standard deep RL process just described. First, we utilize a more sophisticated sampling method during experience replay known as *prioritized experience replay* (PER) (Schaul et al. 2016). In PER, memories are more likely to be sampled if they are deemed more important, i.e., if the loss associated with a memory is not yet known (because the memory has not yet been sampled during replay) or if the loss is large (if the agent was more "surprised" by the memory). Second, we use the double DQN (DDQN) architecture (Hasselt et al. 2016), which employs a "target" DQN for action evaluation and a "primary" DQN for action selection. The target DQN is a clone of the primary DQN that gets updated less often (every M_{update} steps), which helps break the tendency for DQNs to overestimate Q-values. Third, we employ dueling DDQN (D3QN) architecture (Wang et al. 2016), in which the DQN produces an estimate both of the value of the state and of the

relative *advantage* of each action. By producing these values independently (which together combine to yield Q-values), dueling architecture allows the agent to forgo the learning of action-specific values in states in which its decision-making is largely irrelevant. Finally, we use n-step learning (Sutton 1988), which helps to reduce error in learned Q-values. With n-step learning, the rewards r and subsequent states s' stored in an agent's memories are modified: rather than using the state encountered one step into the future from s (s'), the subsequent state stored in the memory is that encountered n-steps into the future ($s^{(n)}$). Similarly, the reward r is replaced by the (properly discounted) sum of the next n rewards, i.e., those accumulated between s and $s^{(n)}$.

The process of choosing which deep RL extensions to implement and values for associated hyperparameters is not unlike traditional heuristic approaches in transportation optimization. While simple heuristics may suffice in simpler problems (see, e.g., Clarke and Wright 1964), more complex problems often demand more complex (meta)heuristics specifically tuned for a particular application (see, e.g., Gendreau et al. 2002). Analogously, off-the-shelf deep RL may be successful in simpler problems (e.g., Karpathy 2016), but success in more complex problems requires enhancements (e.g., Hessel et al. 2018) and is dependent on hyperparameter values (Henderson et al. 2018). The suite of deep RL extensions adopted here and our selection of hyperparameter values (described in §B) are guided by standard values, where available, and ultimately chosen empirically over the course of the work.

4.1.3 Dart.

Our first proposed agent, Dart uses a combination of deep Q-learning and optimization-based logic to choose an action $\mathbf{a} = (a_r, (a_v)_{v\in\mathcal{V}})$ from state $s = (s_t, s_r, s_\mathcal{V})$.

Assigning vehicles to requests. The action a_r pairing a vehicle with the new request employs a deep Q-network. We denote by $\mathcal{A}_{\text{Dart}} = \mathcal{V} \cup \{\emptyset\}$ the action space for Dart's DQN. The DQN takes as input the concatenation of a subset of features from the state s which we represent by $x_{\text{Dart}} = x_t \oplus x_r \oplus x_\mathcal{V}$ ("$\oplus$" is the concatenation operator). These components are defined as follows:

x_t **System time** : the concatenation of the relative time elapsed t/T and a *one-hot vector* indicating the day of the week d (e.g., for zero-indexed d with $d = 0$ corresponding to Monday, $d = 6$ corresponding to Sunday, and d currently equal to Thursday ($d = 3$), the vector $(0, 0, 0, 1, 0, 0, 0)$).

x_r **Request information** : the concatenation of a binary indicator for whether there is a new request $\mathbf{1}_{s_r\neq\emptyset}$; a one-hot vector indicating the location of the request's origin; a one-hot vector indicating the request's destination; and for each vehicle v, the distance v would have to travel to reach the request's origin (scaled by the maximum such distance so values are in $[0, 1]$ (ineligible vehicles are given value 1)).

$x_\mathcal{V}$ **Vehicle information** : the concatenation of x_v for all vehicles $v \in \mathcal{V}$ ($x_\mathcal{V} = x_1 \oplus \cdots \oplus x_V$), where the vehicle-specific x_v is itself the concatenation of the time at which its last non-preemptable job ends (scaled by $1/T$), the charge with which it will finish its last non-preemptable job (scaled by $1/Q$), and a one-hot vector indicating its current location.

One-hot vectors in x_{Dart} that indicate location rely on a discretization of the fleet's operating region, as is common in other dynamic ridehail problems in the literature (e.g., Al-Kanj et al. 2018; Holler et al. 2019). We denote the set of discrete locations (*taxi zones* (TZs)) constituting the region by $\mathcal{L}$.

x_{Dart} is passed to the agent's DQN to produce Q-value predictions. It then uses an ϵ-greedy policy as described in §4.1.1 to choose an $a_r \in \mathcal{A}_{\text{Dart}}$ (infeasible vehicles are ignored). A schematic of Dart's DQN is shown at left in Figure 1.

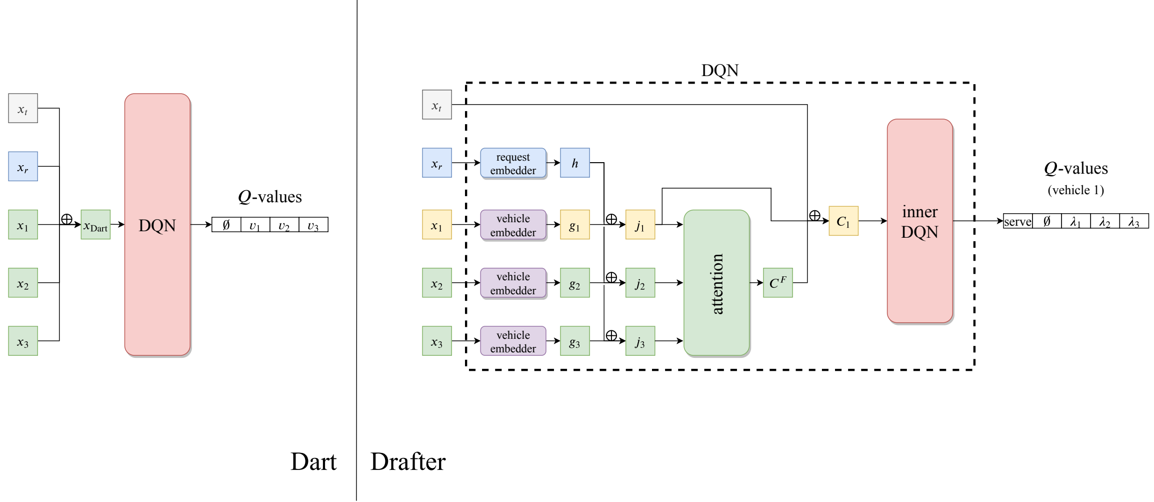

Figure 1: Schematics of the Dart and Drafter agents in the case of three vehicles and three TZs ($V = 3$, $\mathcal{L} = 3$). Elements are concatenated at intersections marked with $\oplus$. The Drafter schematic shows the case where Q-values are being predicted for vehicle 1 (in yellow). Note: in practice, a single forward pass with x_{Drafter} can be used to generate Q-values for all vehicles. We depict the process for a single vehicle here for the sake of clarity.

RNR Actions. For RNR actions, whenever a request has been missed ($s_r \neq \emptyset \wedge a_r = \emptyset$), Dart repositions a stationary vehicle to the CS nearest the origin of the missed request, choosing the vehicle that is closest to this CS. Let c_r^* be the CS nearest the origin of the request; $\widetilde{\mathcal{V}} \subseteq \mathcal{V}$ be the subset of the vehicles that are either idling or charging at a CS ($j_m^{(1)} \in \{0, 1\}$); $f_t(s_v, s_r)$ be the time for vehicle v to reach the origin of the request; and c_v^* be the CS nearest vehicle v. Then Dart's setting of RNR actions can be stated as

$$a_v = \begin{cases} c_r^* & s_r \neq \emptyset \wedge a_r = \emptyset \wedge v = \arg\min_{v \in \widetilde{\mathcal{V}}} f_t(s_v, s_r) \\ \emptyset & j_m^{(1)} \neq \emptyset \\ c_v^* & \text{otherwise.} \end{cases} \tag{6}$$

In equation (6), the first case specifies the repositioning of the nearest eligible vehicle to the CS nearest the origin of a missed request. The second case specifies the NOOP action for all vehicles for which it is feasible (i.e., those with a valid current job), and the third case specifies that the remaining vehicles (which do not have a valid current job) are assigned to reposition to the nearest CS.

4.1.4 Drafter.

From a state s, the Drafter agent uses a DQN to choose a vehicle to serve new requests a_r and to provide RNR instructions a_v for each vehicle $v \in \mathcal{V}$. Whereas the number of unique actions under the control of Dart's DQN is $V + 1$, for Drafter it is $(V + 1) \times (C + 1)^V$. For non-trivial fleet sizes V, this is intractably large. In response, we employ a multi-agent reinforcement learning (MARL) framework which avoids this intractability by making separate, de-centralized Q-value predictions for each vehicle. This reduces the number of actions under the control of Drafter's DQN to $C + 2$, corresponding to the vehicle-specific actions of relocating to each station $c \in \mathcal{C}$, serving the request, and doing nothing. We further reduce the number of actions by using the discretization into TZs $\mathcal{L}$ described in §4.1.3, yielding the action

space $\mathcal{A}_{\text{Drafter}} = \{\text{serve}, \emptyset\} \cup \mathcal{L}$ for Drafter's DQN. These actions correspond respectively to serving the request, doing nothing, and relocating to each TZ $\lambda \in \mathcal{L}$. For vehicle v, we define its DQN input by $x^v_{\text{Drafter}} = (x_t, x_r, x_v, x_{\mathcal{V}})$, where components are as defined for Dart in §4.1.3 and the redundant x_v is included just to indicate the vehicle for which we are generating Q-values. With vehicle input x^v_{Drafter}, the DQN makes Q-value predictions $Q(x^v_{\text{Drafter}}, a)$ for $a \in \mathcal{A}_{\text{Drafter}}$. These predictions are collected for all vehicles and used to make a centralized decision. A schematic of Drafter's DQN is shown at right in Figure 1.

DQN and attention mechanism. Let us consider the prediction of Q-values for a vehicle $v^\star$. In the example in Figure 1, $v^\star = v_1$. We now also incorporate into the DQN an *attention mechanism* (Mnih et al. 2014) to improve the agent's ability to distinguish the most relevant vehicles in the current state. We expand on the attention mechanism used in Holler et al. (2019) by incorporating information about the current request, which is likely to influence vehicles' relevancy. Details of our attention mechanism can be found in the appendix, §B. The attention is used to provide an alternative representation (an *embedding*) of each vehicle $g_v \in \mathbb{R}^l$ and the request $h \in \mathbb{R}^m$, as well as a description of the fleet $C^F \in \mathbb{R}^n$ known as the *fleet context* (here $l = n = 128$, $m = 64$; see §B). These components are used to form the vector $C_{v^\star} = C^F \oplus g_{v^\star} \oplus h \oplus x_t$ which is fed to an inner DQN. This inner DQN, which has a structure similar to Dart's DQN, outputs Q-values $Q(x^{v^\star}_{\text{Drafter}}, a)$ for $a \in \mathcal{A}_{\text{Drafter}}$. The process is repeated for each $v \in \mathcal{V}$, with each vehicle using the same attention mechanism and inner DQN (i.e., all weights are shared).

Drafter is made scalable both through its use of an MARL approach as described above, and also by the attention mechanism. While the MARL approach ensures that the output of the DQN does not depend on the size of the fleet, the attention mechanism ensures the same for the input. More specifically, the attention mechanism makes it possible to predict Q-values (via the inner DQN) using a representation of the fleet C^F whose size is not determined by the number of vehicles in the fleet. This structure allows Drafter to be applied to instances of a different size than those on which it was trained. For example, given a Drafter agent trained to operate a fleet of size V, if a new vehicle is purchased and added to the fleet, or if a vehicle malfunctions and must be temporarily removed, Drafter is capable of continuing to provide service for this modified fleet of size $V' \neq V$. We assess the scalability of the Drafter agent in our computational experiments, described in §6.

Centralized decision-making. Given Q-value predictions for all vehicles, Drafter's centralized decision-making proceeds as follows. Let $\bar{\mathcal{V}}_{\text{Drafter}}$ be the set of vehicles for whom the service action has the largest Q-value. To serve the request, we choose the vehicle in $\bar{\mathcal{V}}_{\text{Drafter}}$ with the largest such Q-value; this vehicle then receives NOOP for its RNR action. The remaining vehicles perform the RNR action with the largest Q-value, with NOOP prioritized in the event of a tie (ties between repositioning actions are broken arbitrarily). Some consideration must be given to repositioning assignments a_v, as the DQN produces outputs in $\mathcal{L}$ (TZs), while actions a_v take values in $\mathcal{C}$ (stations). Given a repositioning instruction to some TZ $\lambda^\star \in \mathcal{L}$ for vehicle v, we set a_v to the station in $\lambda^\star$ that is nearest to the vehicle.

Given an action $\mathbf{a} = (a_r, (a_v)_{v \in \mathcal{V}})$, define a^v_{Drafter} to be the action in Drafter's action space $\mathcal{A}_{\text{Drafter}}$ assigned to vehicle v:

$$a^v_{\text{Drafter}} = \begin{cases} \text{serve} & a_r = v \\ L(a_v) & \text{otherwise}, \end{cases} \tag{7}$$

where $L(a_v)$ represents the TZ that contains station a_v.

Experience Replay. As described in §4.1.2, Drafter saves state-transition memories $(s, \mathbf{a}, r, s^{(n)})$ at each step which are later used to train its DQN via prioritized experience replay. Let $x^{v(n)}_{\text{Drafter}}$ be the DQN input for vehicle v from state $s^{(n)}$. Because Drafter makes V predictions at each step, our sample of M_{batch} memories leads to an effective sample size of $V * M_{\text{batch}}$, where each memory $(s, \mathbf{a}, r, s^{(n)}) = (x^v_{\text{Drafter}}, a^v_{\text{Drafter}}, r, x^{v(n)}_{\text{Drafter}})_{v\in\mathcal{V}}$. Note that the reward r is shared equally among all vehicles during training, with each vehicle receiving the full amount. This was chosen to encourage cooperation among the fleet. Thus, for Drafter, the difference that forms the basis of its loss function (c.f. equation (5)) is

$$r + \gamma \max_{a' \in \mathcal{A}_{\text{Drafter}}} Q(x^{v(n)}_{\text{Drafter}}, a') - Q(x^v_{\text{Drafter}}, a^v_{\text{Drafter}}). \tag{8}$$

4.2 Benchmark Policies

We implement two policies as benchmarks against which to assess the performance of our Dart and Drafter agents. First, in §4.2.1 we describe the *Random* policy that chooses actions randomly, effectively serving as a lower bound. Then in §4.2.2 we describe the *Reopt* policy, a competitive benchmark from the ridehailing literature that leverages reoptimization to assign vehicles to requests and to reposition the vehicles in the fleet (Alonso-Mora et al. 2017).

4.2.1 Random.

The *Random* policy chooses a vehicle to serve new requests (a_r) uniformly from the eligible vehicles in $\mathcal{V} \cup \{\emptyset\}$ and chooses repositioning assignments a_v uniformly from the eligible locations in $\mathcal{C} \cup \{\emptyset\}$ for each $v \in \mathcal{V}$.

4.2.2 Reopt.

Our Reopt agent employs the reoptimization approach introduced in Alonso-Mora et al. (2017), solving two mixed integer programs (MIPs) in every epoch to inform its decisions. This agent is inherently different from the other three proposed in this section. Whereas decision epochs for Dart, Drafter, and Random are triggered by the arrival and completion of requests, the Reopt agent is afforded a decision epoch at regular intervals (governed by the parameter T_{reopt}; e.g., every 30 seconds). This difference requires modifications to the problem model defined in §3.

Required model modifications. The RNR actions from which the Reopt agent may choose are identical to those for the other agents. However, the service action a_r now takes a different form. Because the Reopt agent pools requests for T_{reopt} time in between decision epochs, there may now be more than one request to which the agent must respond. Thus, a_r is now a set of tuples (v, r), corresponding to matchings between a vehicle $v \in \mathcal{V} \cup \emptyset$ with a request $r \in s_{\mathcal{R}}$, where $s_{\mathcal{R}}$ is the set of requests that arrived in the last T_{reopt} time. A rejected request is denoted by the matching $(\emptyset, r)$. Each request $r \in s_{\mathcal{R}}$ has origin and destination attributes as described in §3, however, we now also include the time at which it was received t_r.

Assigning vehicles to requests. To determine how to assign vehicles to requests, Reopt solves a MIP minimizing the total customer waiting time. Let $\omega_{v,r}$ be the waiting time of customer r if vehicle v is assigned to serve it. If the current time is s_t, the vehicle's state is s_v, the request's arrival time is t_r, and (as in §3) the time for the vehicle to arrive to the origin of the request from its current position is $f_t(s_v, r)$; then we have $\omega_{v,r} = (s_t - t_r) + f_t(s_v, r)$. We define two sets of binary variables: $m_{v,r}$, which

take value 1 if request r is served by vehicle v; and ψ_r, which take value 1 if request r is served by any vehicle (is not rejected). These variables are otherwise 0. Let $\mathcal{M}$ be the set of (v, r) matchings that are either energy- or time-infeasible, as defined in §3. The MIP to determine the assignment of vehicles to new requests is then

$$\text{minimize} \quad \sum_{r \in s_{\mathcal{R}}} \sum_{v \in \mathcal{V}} \omega_{v,r} m_{v,r} - \sum_{r \in s_{\mathcal{R}}} B\psi_r \tag{9}$$

$$\text{subject to} \quad \sum_{r \in s_{\mathcal{R}}} m_{v,r} \leq 1, \quad \forall v \in \mathcal{V} \tag{10}$$

$$\sum_{v \in \mathcal{V}} m_{v,r} = \psi_r, \quad \forall r \in s_{\mathcal{R}} \tag{11}$$

$$m_{v,r} = 0, \quad \forall (v, r) \in \mathcal{M} \tag{12}$$

$$\psi_r, m_{v,r} \in \{0, 1\} \tag{13}$$

The second term in the objective, equation (9), contains the large positive constant B ensuring that the model does not unnecessarily ignore requests. Constraints (10) ensure that each vehicle is assigned to at most one request. Constraints (11) serve both to ensure that at most one vehicle is assigned to serve each request and to fix the ψ_r variables, denoting whether requests will be served. Constraints (12) disallow infeasible matchings, and constraints (13) provide variable definitions.

Following the solution to this MIP, we build the set of vehicle-request matchings a_r as follows. Any request r for which $\psi_r = 0$ is unserved and receives the matching $(\emptyset, r)$. For the remaining requests (with $\psi_r = 1$), we have matchings (v, r) corresponding to the non-zero $m_{v,r}$ variables.

RNR actions. Per the approach in Alonso-Mora et al. (2017), after assigning vehicles to requests, Reopt then solves another MIP to reposition the vehicles in the fleet. This MIP minimizes the time to reposition idling vehicles so as to be close to the origin of any missed requests. Alonso-Mora et al. motivate this approach with the assumptions that missed requests may request again, that more requests are likely to occur in the same area where not all requests can be satisfied, and that idling vehicles are in areas with insufficient demand. We note that our Dart agent uses a modified version of this approach for its RNR actions as well.

Let $\widetilde{\mathcal{V}} \subseteq \mathcal{V}$ be the subset of the fleet that is either idling or charging at a station ($j_m^{(1)} \in \{0, 1\}$) and was not just assigned to serve a new request in the assignment MIP (equations (9)-(13)), and let $\widetilde{s}_{\mathcal{R}} \subseteq s_{\mathcal{R}}$ be the subset of the new requests which were rejected (received a matching $(\emptyset, r)$). For each request $r \in \widetilde{s}_{\mathcal{R}}$, define c_r^* to be the station nearest its origin. Define analogous coefficients $\widetilde{\omega}_{v,r}$ to be the travel time for vehicle $v \in \widetilde{\mathcal{V}}$ to reach c_r^*, and let $\widetilde{\mathcal{M}}$ be the analogous set of energy-infeasible repositioning assignments. We define binary variables $\widetilde{m}_{v,r}$ which take value 1 if vehicle v is repositioned to station c_r^* and are 0

otherwise. Then the MIP to determine fleet repositioning is as follows:

$$\text{minimize} \quad \sum_{r \in \widetilde{s}_{\mathcal{R}}} \sum_{v \in \widetilde{\mathcal{V}}} \widetilde{\omega}_{v,r} \widetilde{m}_{v,r} \tag{14}$$

$$\text{subject to} \quad \sum_{r \in \widetilde{s}_{\mathcal{R}}} \sum_{v \in \widetilde{\mathcal{V}}} \widetilde{m}_{v,r} = \min\left(\left|\widetilde{\mathcal{V}}\right|, \left|\widetilde{s}_{\mathcal{R}}\right|\right) \tag{15}$$

$$\sum_{r \in \widetilde{s}_{\mathcal{R}}} \widetilde{m}_{v,r} \leq 1, \quad \forall v \in \widetilde{\mathcal{V}} \tag{16}$$

$$\sum_{v \in \widetilde{\mathcal{V}}} \widetilde{m}_{v,r} \leq 1, \quad \forall r \in \widetilde{s}_{\mathcal{R}} \tag{17}$$

$$\widetilde{m}_{v,r} = 0, \quad \forall (v,r) \in \widetilde{\mathcal{M}} \tag{18}$$

$$\widetilde{m}_{v,r} \in \{0,1\} \tag{19}$$

Constraint (15) ensures that we provide the maximum number of repositioning assignments: either we provide an assignment for all eligible vehicles, or we provide an assignment for all missed requests. Constraint set (16) ensures that each eligible vehicle is given at most one repositioning assignment, and constraint set (17) ensures that each missed request is assigned to be the repositioning destination for at most one vehicle. Constraints (18) disallow infeasible matchings, and constraints (19) provide variable definitions.

Given a solution to this MIP, Reopt makes vehicle assignments $a_v = c_r^*$ for any non-zero variable assignments $\widetilde{m}_{v,r}$. Any vehicle without such an assignment simply receives the NOOP action $a_v = \emptyset$ unless not allowed, in which case the vehicle is instructed to reposition to its nearest station. Formally,

$$a_v = \begin{cases} c_r^* & \widetilde{m}_{v,r} = 1 \\ \emptyset & j_m^{(1)} \neq \emptyset \\ c_v^* & \text{otherwise,} \end{cases} \tag{20}$$

where c_v^* is the closest station to vehicle v.

We offer Reopt agent as a benchmark for a good, scalable dynamic policy. In general, reoptimization approaches offer several potential advantages, foremost among them the ability to consolidate multiple customer requests into a single decision-making epoch. This effectively increases the amount of information with which the fleet operator can make decisions. As such, a good policy that pools requests should be at least as good as a policy that serves requests immediately.

5 Dual Bounds

Assessing policy quality is hampered by the lack of a strong bound on the value of an optimal policy, a dual bound. Without an absolute performance benchmark it is difficult to know if a policy's performance is "good enough" for practice or if additional research is required. Here, we offer two dual bounds. First is a bound on the value of a policy that serves all requests (§5.1), and the second is the value of an optimal policy with perfect information, i.e., the performance achieved via a clairvoyant decision maker (§5.2).

5.1 Serve-All Bound.

We first consider the *Serve-all* (SA) dual bound, which is a bound on the value of a policy that serves all requests. To establish this bound, we assume that the policy is able to serve all requests without

incurring any setup travel costs. That is, we can simply sum the value of all observed requests, minus the travel costs incurred to serve them: $C_B + (C_D - C_T) \cdot d(s_r)$, where, as in §3, C_B is the base revenue earned when serving a request, C_D is the per-distance revenue earned when serving a request, C_T is the per-distance travel costs incurred, and $d(s_r)$ is the distance traveled serving the request s_r. We know that, given a fleet of sufficient size (and assuming $C_D \geq C_T$ and $C_B \geq 0$), an optimal policy would choose to serve all requests, because policies seek to maximize the sum of rewards, which here are non-negative and incurred only by serving requests. The gap between the SA dual bound and the best policy is a reflection of vehicles' extraneous traveling and the adequacy of fleet size relative to demand and can serve as justification for a ridehailing company to invest (or not) in additional vehicles.

5.2 Perfect Information Bound.

In practice, setup and some additional travel costs are unavoidable, and it is likely that not all requests can be feasibly served. As a result, the SA bound is expected to be loose. In an attempt to establish a tighter dual bound, we consider the value of an optimal policy under a *perfect information* (PI) relaxation. Under the PI relaxation, the agent is clairvoyant, aware of all uncertainty a priori. In the E-RPC, to have access to PI is to know in advance all details regarding requests: their origins, destinations, and when they will arise. Denote such a set of known requests by $\mathcal{R} \in \mathrm{P}$, where P is the set of all request sets.

In the absence of uncertainty, we can rewrite the objective function as

$$\max_{\pi \in \Pi} \mathbb{E} \left[\sum_{k=0}^{K} C(s_k, X_k^\pi(s_k)) \middle| s_0 \right] = \mathbb{E} \left[\max_{\pi \in \Pi} \sum_{k=0}^{K} C(s_k, X_k^\pi(s_k)) \middle| s_0 \right]. \tag{21}$$

Notice that the *perfect information problem* (right-hand side of equation (21)) can be solved with the aid of simulation. We may rely on the law of large numbers — drawing random realizations of uncertainty (request sets $\mathcal{R} \in \mathrm{P}$), solving the inner maximization for each, and computing a sample average — to achieve an unbiased and consistent estimate of the true objective value. This value is the *perfect information bound*. It remains to solve the inner maximization.

In the absence of uncertainty that results from having access to PI, the inner maximization can be solved deterministically: with all information known upfront, no information is revealed to an agent during the execution of a policy. As a result, there is no advantage in making decisions dynamically (step by step) rather than statically (making all decisions at time 0). This permits the use of an exact solution via math programming, which we pursue using a *Benders-based branch-and-cut* algorithm in which at each integer node of the branch-and-bound tree of the *master problem*, the solution is sent to the *subproblem* for the generation of Benders cuts. Here, the master problem is responsible for assigning vehicles to requests in a time-feasible manner, and the subproblem is responsible for ensuring the energy feasibility of these assignments. Additionally, the master problem produces an estimate of the total costs incurred through a particular assignment, and the subproblem refines this cost estimate. The use of the Benders-based decomposition enables the solution of larger PI problems than would otherwise be feasible. We discuss the master problem in more detail in §5.2.1, the subproblem and the generation of cuts in §5.2.2, and comment on the bound's tractability in §5.2.3.

5.2.1 Master problem.

The master problem, responsible for time-feasible assignments of customer requests to vehicles and providing an estimate of the total profit earned, is the MIP defined by equations (22)-(28). In it, requests

are represented as nodes in a directed graph $\mathcal{G}$. Request nodes i and j are connected by a directed arc (i, j) when a vehicle can feasibly serve request j after request i (ignoring energy requirements). The graph $\mathcal{G}$ also contains a dummy node for each vehicle, representing the location from which it initially departs. These dummy nodes are connected to request nodes for which the initial assignment of vehicles to requests is time-feasible. The problem involves choosing arcs in $\mathcal{G}$, starting from vehicles' dummy nodes, that form (non-overlapping) paths for the vehicles which indicate the sequence of requests that vehicles will serve. If a request i is a member of some vehicle's path, it contributes c_i to the objective function, the revenue that its service generates. Our objective also includes the travel costs incurred along the paths that the vehicles travel in serving their assigned requests.

In the master problem, $\mathcal{R}$ is the set of all requests (known a priori given PI), $\mathcal{V}$ is the set of vehicles, c_i is the revenue earned by serving request i, h_i is a binary variable taking value 1 if request i is assigned to any vehicle, y_{vi} is a binary variable taking value 1 if job i is the first request assigned to vehicle v, x_{ij} is a binary variable taking value 1 if a vehicle is assigned serve request j immediately after request i, z_i is a continuous non-negative variable equal to the time at which a vehicle arrives to pick up request i, t_i^r is the time at which request i begins requesting service, w^* is the maximum amount of time a customer may wait between when they submit their request and when a vehicle arrives, t_i^p (d_i^p) is the travel time (distance) between the origin and destination of request i, and t_{ij}^s (d_{ij}^s) is the travel time (distance) between the destination of request i and the origin of request j. For vehicles, t_v^p is equal to the earliest time at which they can depart from their initial locations, and t_{vi}^s (d_{vi}^s) is equal to the travel (distance) time between their initial locations and the origin of request i. We formally define the master problem as

$$\text{maximize} \quad \sum_{i \in \mathcal{R}} c_i h_i - C_T \left(\sum_{v \in \mathcal{V}} \sum_{i \in \mathcal{R}} d_{vi}^s y_{vi} + \sum_{i \in \mathcal{R}} \sum_{j \in \mathcal{R} \setminus \{i\}} d_{ij}^s x_{ij} + \sum_{i \in \mathcal{R}} d_i^p h_i \right) - \theta_m \tag{22}$$

$$\text{subject to} \quad \sum_{v \in \mathcal{V}} y_{vi} + \sum_{j \in \mathcal{R} \setminus \{i\}} x_{ji} \geq h_i, \quad \forall i \in \mathcal{R} \tag{23}$$

$$\sum_{j \in \mathcal{R} \setminus \{i\}} x_{ij} \leq h_i, \quad \forall i \in \mathcal{R} \tag{24}$$

$$\sum_{j \in \mathcal{R}} y_{vj} \leq 1, \quad \forall v \in \mathcal{V} \tag{25}$$

$$z_i \geq t_i^r + (t_v^p + t_{vi}^s - t_i^r) y_{vi}, \quad \forall i \in \mathcal{R}, \forall v \in \mathcal{V} \tag{26}$$

$$z_i - z_j \geq t_i^r - t_j^r - w^* + (t_j^p + t_{ji}^s - t_i^r + t_j^r + w^*) x_{ji}, \quad \forall i, j \in \mathcal{R}\, (i \neq j) \tag{27}$$

$$h_i, x_{ij}, y_{vi} \in \{0, 1\};\ t_i^r \leq z_i \leq t_i^r + w^*;\ \theta_m \geq 0 \tag{28}$$

The objective (22) maximizes profit: its first term captures the revenue earned, while the remaining terms account for costs. In the objective, the first two terms in parentheses capture the distance the vehicles travel in setting up for (traveling between) requests, and the third term in parentheses captures the distance traveled by the vehicles while serving the requests. In its last term, the objective also includes the continuous variable θ_m. This variable is controlled by optimality cuts generated in the subproblem, effectively serving to correct the master problem's underestimation of travel costs. While the master problem's objective accounts for the minimum incurred travel costs (those incurred by traveling request sequences directly), there will likely be additional costs incurred as vehicles travel to and from stations. These station visits are required both for vehicles to restore their charge and to idle between requests, since they are not allowed to do so at request locations. Thus, intuitively, θ_m represents the costs incurred during these required station detours.

While θ_m is ultimately controlled by the subproblem, we can facilitate the solution of the PI problem by improving the master problem's estimation of these costs. To do so, we consider the minimum required detours between pairs of requests. For requests i and j, let k be the station requiring the minimum detour between i's destination and j's origin, and let t^d_{ij} (d^d_{ij}) be the time (distance) required to travel from the destination of request i to station k to the origin of request j. Then we define the minimum detour distances between requests

$$\ell_{ij} = \begin{cases} d^d_{ij} & t^r_i + w^* + t^p_i + t^d_{ij} \leq t^r_j \\ 0 & \text{otherwise,} \end{cases} \tag{29}$$

and then apply the following condition on θ_m:

$$\theta_m \geq \sum_{i \in \mathcal{R}} \sum_{j \in \mathcal{R} \setminus \{i\}} C_T \ell_{ij} x_{ij}. \tag{30}$$

In equation (29) we specify that if there is sufficient time for a vehicle to perform a detour between requests i and j then it must do so, and in equation (30) we account for these additional travel costs by raising the lower bound on θ_m.

For the remainder of the master problem, equation (23) manages the binary assignment variable h_i, requiring that it be included in some vehicle's path to take value 1. Similarly, equation (24) manages the variables for the outgoing arcs from request i, forcing them to take value 0 if the request has not been assigned to a vehicle. Equation (25) ensures that each vehicle has at most one request assigned to be its first. Equation (26) sets a lower bound for the time at which vehicles can arrive to their initial requests, and equation (27) sets a lower bound for the time at which vehicles' can arrive to subsequent requests. Finally, equation (28) defines variables scopes' and bounds the earliest and latest possible start times for requests z_i.

As mentioned, we only connect request nodes i and j via directed arc (i, j) if it is time-feasible to serve request j after request i; that is, if $t^r_i + t^p_i + t^s_{ij} \leq t^r_j + w$. While energy-feasibility is ultimately ensured by the subproblem, we can again facilitate in the master problem by eliminating additional arcs in $\mathcal{G}$ using known energy consumptions to perform stronger feasibility checks. Let ρ be the maximum rate at which vehicles acquire energy when recharging, q^p_i be the energy required to travel from the origin to the destination of request i, q^s_{ij} be the energy required to travel from the destination of request i to the origin of request j, q^d_i be the energy required to travel from the destination of request i to the nearest charging station, and q^o_i be the energy required to travel to the origin of request i from its nearest charging station. Then for arc (i, j) to exist, we require that $t^r_j + w - (t^r_i + t^p_i + t^s_{ij}) \geq \frac{1}{\rho} \max\{0, (q^p_j + q^d_j - (Q - q^o_i - q^p_i))\}$, which states that the maximum down time between i and j (left-hand side) be sufficient to accommodate any recharging that must occur between these requests (right-hand side). We provide similar feasibility checks for the arcs (v, i) connecting vehicle dummy nodes to requests. Specifically, the simpler time-feasible check requires that vehicle v can arrive to request i in time: $t^p_v + t^s_{vi} \leq t^r_i + w^*$. In consideration of energy requirements, we ensure $t^r_i + w^* - (t^p_v + t^s_{vi}) \geq \frac{1}{\rho} \max\{0, q^s_{vi} + q^p_i + q^d_i - q^p_v\}$, which states that the time for the vehicle to perform any required charging (right-hand side) cannot be greater than the available time before it must arrive to request i. Finally, we force the underlying graph to be acyclic, meaning we prohibit pairs of requests i, j such that $(t^r_i + t^p_i + t^s_{ij} < t^r_j + w^*) \wedge (t^r_j + t^p_j + t^s_{ji} < t^r_i + w^*)$.

5.2.2 Subproblem.

A solution to the master problem is a sequence of request assignments $r_v = (r^1_v, r^2_v, \ldots)$ for each vehicle $v \in \mathcal{V}$. r^1_v is taken to be the element in the singleton $\{j \,|\, y_{vj} = 1\} \cup \emptyset$; subsequent entries r^i_v are similarly

elements in singletons $\{j|x_{r_v^{i-1}j} = 1\} \cup \emptyset$ (r_v terminates with the first null element). Given sequences for each vehicle, the subproblem must ensure they are energy feasible. To do so, we use a modified version of *frvcpy* (Kullman et al. 2020), which implements the labeling algorithm developed by Froger et al. (2019) to solve the Fixed Route Vehicle Charging Problem (FRVCP) (Montoya et al. 2017). The FRVCP entails providing charging instructions (where to charge and to what amount) for an electric vehicle traversing a fixed sequence of customers. The algorithm takes as input a sequence like r_v and, if successful, terminates with the minimum duration path through the sequence that includes instructions specifying at which charging stations to recharge and to what amount. When unsuccessful, the labeling algorithm returns the first unreachable node in the sequence. In the original implementation customers did not have time constraints as they do here, where vehicles are required to arrive to requests in the window $[t_i^r, t_i^r + w^*]$. We further discuss the algorithm and our modifications to it to accommodate this difference in the appendix, §C.

Unsuccessful termination of the algorithm for a sequence r_v indicates that the sequence cannot be traversed energy-feasibly. To remove it from the search space, we add the following *feasibility cut* to the master problem:

$$y_{v,r_v^1} + \sum_{i=1}^{j^\star-1} x_{r_v^{i-1},r_v^i} < j^\star, \tag{31}$$

where $j^\star \leq |r_v|$ is the index of the first unreachable request node in r_v to which the algorithm was unable to feasibly extend a label. This cut (31) enforces that not all variables defining the subsequence $(r_v^1, \ldots, r_v^{j^\star})$ be selected.

When the algorithm terminates successfully, we add *optimality cuts* that correct the master problem's estimation of the costs incurred in serving the sequence r_v. Specifically, let $C(r_v)$ represent the master problem's estimation of the costs incurred by traveling sequence r_v,

$$C(r_v) = C_T \left(d^s_{v,r_v^1} + \sum_{i=1}^{|r_v|-1} d^s_{r_v^i,r_v^{i+1}} + \sum_{i=1}^{|r_v|} d_i^p \right), \tag{32}$$

and let $C^*(r_v)$ represent the actual total cost of traveling sequence r_v, as determined by the solution of the labeling algorithm. Then we add optimality cuts to the master problem that account for the difference:

$$\theta_m \geq (C^*(r_v) - C(r_v)) \left(\left(y_{v,r_v^1} + \sum_{i=1}^{|r_v|-1} x_{r_v^{i-1},r_v^i} \right) (|r_v| - 1) \right). \tag{33}$$

These cuts work by ensuring that if the master problem selects sequence r_v, then it must account for its true travel costs by setting $\theta_m \geq C^*(r_v) - C(r_v)$. Otherwise, the right-hand side of equation (33) is at most 0, which is redundant given the non-negativity constraint on θ_m, equation (28).

5.2.3 Tractability of the PI Bound.

The PI bound is often computationally intractable to obtain. This is because the estimation of the expected value with perfect information entails repeated solutions to the inner maximization of equation (21), a challenging problem despite the absence of uncertainty. Even if the Benders-based method of §5.2.1 and 5.2.2 does not return an optimal solution for a given realization of uncertainty, we may still get a valid bound by using the best (upper) bound produced by the solver (Gurobi v8.1.1). The solver's bound serves as a bound on the value of an optimal policy with PI, and therefore also as a bound on an

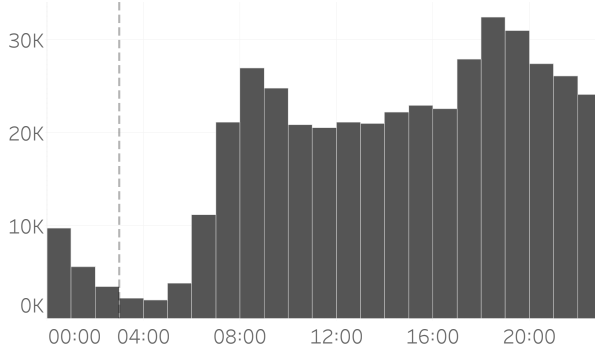

Figure 2: Mean number of requests by hour in Manhattan on a business day in 2018. The dashed vertical line indicates episodes' 03:00 start time.

optimal policy. This mixed bound, effectively combining both the solver's relaxation and an information relaxation, is weaker than a bound based on the information relaxation alone. However, we show in computational experiments that it still provides value.

6 Computational Experiments

To test the solution methods proposed in §4, we perform computational experiments modeled after business-day ridehailing operations on the island of Manhattan in New York City. We describe the experimental setup in §6.1, present our results in §6.2, and offer a brief discussion of the results in §6.3.

6.1 Experimental Setup

We generate problem instances (equivalently, *episodes* over which the agents act) using data publicly available for the island of Manhattan in New York City. New York City Taxi & Limousine Commission (2018) provide a dataset consisting of all ridehail, Yellow Taxi, and Green Taxi trips taken in 2018. We filter the data to only those trips that originate and terminate in Manhattan and those that occur on business days ($d \in \{0, 1, 2, 3, 4\} = \{\text{Mon, Tues, Weds, Thurs, Fri}\}$, excluding public holidays). The average daily profile for these data is shown in Figure 2. Based on this profile, we set the episode length T to 24 hours, with episodes beginning and ending at 3:00 am, as this time corresponds to a natural lull in demand from the previous day and gives agents time to perform proactive recharging before the morning demand begins. Data for each trip includes the request's pickup time as well as the location of its origin and destination. Requests' origins and destinations are given by their corresponding *taxi zones*, an official division of New York City provided by the city government (NYC OpenData 2019) that roughly divides the city into neighborhoods. We use this as the basis for our geographical discretization $\mathcal{L}$ as described in §4, resulting in $|\mathcal{L}| = 61$ TZs for Manhattan (see Figure 3). Coordinates $((o_x, o_y), (d_x, d_y))$ for requests' exact origins and destinations are drawn randomly from inside their corresponding TZs.

We fix the set of stations $\mathcal{C}$ to all CSs currently available or under construction in Manhattan at the time of writing, as listed by National Renewable Energy Laboratory (2019), yielding a set of $|\mathcal{C}| = 302$ CSs (blue marks in left map of Figure 3). We assume that all CSs dispense energy at a constant rate of 72kW, equal to that of a Tesla urban supercharger (Tesla 2017). Vehicles' batteries have similar traits to that of a mid-range Tesla Model 3, with a capacity of 62 kWh and an energy consumption of 0.15 kWh/km. Further, we assume vehicles travel at a speed of approximately 13 km/hr (8 mi/hr), incur

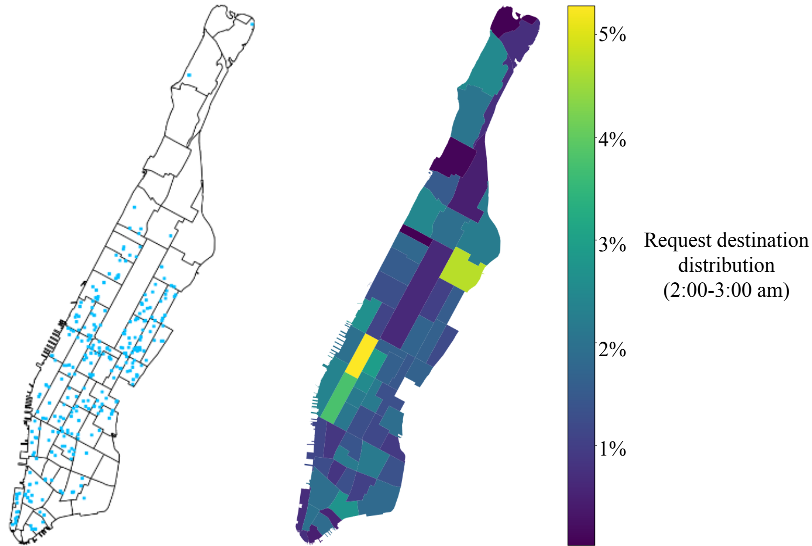

Figure 3: (Left) The island of Manhattan divided into taxi zones with charging station locations shown in blue. (Right) Distribution of requests' desintations by taxi zone during episodes' last hour (02:00-03:00).

travel costs of \$0.53/km (Bösch et al. 2018), and, when serving a request, receive a fixed reward of C_B = \$10.75 and distance-dependent reward of C_D = \$4.02/km, under the assumption of Euclidean distances. We set the maximum time that customers are willing to wait for a vehicle to be $w^* = 5$ minutes. These instance parameters are summarized in Table 2.

Vehicles' initial locations are determined by drawing a TZ randomly according to the distribution of drop-off locations during the final hour of the previous day's operation (2:00-3:00am). See right map in Figure 3. We then choose a charging station uniformly from within the TZ (TZs without CSs are excluded). Vehicles' initial charges are drawn uniformly from $[0, Q]$. To randomly generate a set of R requests for an instance, we first sample a business day from 2018 then draw R requests that occurred on that day.

We consider a primary set of problem instances with $R = 1400$ requests and $V = 14$ vehicles (we will often refer to these instances by their R/V ratio, 1400/14). The number of requests was chosen such that the mean number of requests per hour during the busiest time of day (between 6:00-7:00pm, see Figure 2) is approximately 100. The number of vehicles in this set was then derived from a scaling of the ratio $R^\star/V^\star$, where $R^\star = 449,121$ is the total number of trip requests (taxi and ridehail) served on an average business day in Manhattan in 2018, and $V^\star = 4,400$ is an approximation of the number of simultaneously active Yellow Taxis in Manhattan at any given time in 2018 (see data aggregations in, e.g., Schneider 2019). Experiments on our 1400/14 instances include results for all agents as well as the SA and PI bounds. The agents are evaluated over a set of 200 episodes, while the bounds — given the high computational demand for the PI bound — are established using an average over 30 episodes. Prior to evaluation, the Dart and Drafter agents are first trained on a separate set of 750 episodes. We note that the PI bound cannot be solved to optimality for these instances, so the values reported correspond to the mixed version of the bound as described in §5.2.3.

In addition to the 1400/14 instances, we also consider instances two, five, and ten times larger (2800/28, 7000/70, and 14000/140) to assess Drafter's ability to scale and generalize. That is, we

Table 2: Instance parameters.

Parameter	Value
Episode length T	24 hrs
Episode start time	3:00 am
Number of Taxi Zones $\|\mathcal{L}\|$	61
Number of charging stations $\|\mathcal{C}\|$	302
CS charging rate	72 kW
Vehicle battery capacty	62 kWh
Vehicle energy consumption	0.15 kWh/km
Vehicle speed	13 km/hr
Travel costs C_T	\$0.53/km
Base fare C_B	\$10.75
Per-distance fare C_D	\$4.02/km
Max customer waiting time w^*	5 minutes

evaluate the Drafter agent directly on 200 instances of each size without any additional training on these instances — it simply uses its DQN as trained on the 1400/14 instances. We compare the Drafter agent to the Random and Reopt agents. Dart is absent from this analysis as its scalability is inherently handicapped, with its DQN's output ($\mathcal{A}_{\text{Dart}}$) dependent on the number of vehicles in the instance. This is in contrast to Drafter, for which the size of its DQN output ($\mathcal{A}_{\text{Drafter}}$) is indifferent to the size of the fleet. Details on the hyperparameters used in the training of the Dart and Drafter agents for all instance size classes are provided in §B. We use a reoptimization period of $T_{\text{reopt}} = 30$ seconds for the Reopt agent.

6.2 Results

We begin with a comparison of our proposed dual bounds in §6.2.1. We then describe agents' performance on the instance sets, first discussing the results on our primary 1400/14 instances in §6.2.2, followed by scalability tests in §6.2.3.

6.2.1 Comparison of Dual Bounds.

The first proposed dual bound, SA, is simple to compute but may be loose when not all requests can be feasibly served. In contrast, the PI bound promises to be tighter, but it is significantly more challenging to compute. Here, we aim to quantify the improvement in the bound afforded by the additional computation for the PI bound.

Intuitively, with a larger fleet size V, more requests can be served. At some V, a policy with PI should be able to serve all requests, so the PI and SA bounds should converge. Conversely, with decreasing fleet size, even with PI, it should become impossible to serve all requests, so the bounds will diverge. We test this intuition in an experiment comparing the value of these two bounds as a function of fleet size. The results are summarized in Figure 4. We begin with instances of size 1400/14, for which (as shown in §6.2.2) the gap between the PI and SA bounds is large. We then increment the fleet size by 2 ($V+ = 2$), and resolve 15 episodes of the 1400/V instances. We find that the results confirm intuition – the gap between the bounds decreases exponentially with increasing fleet size, reaching near equivalence (2%

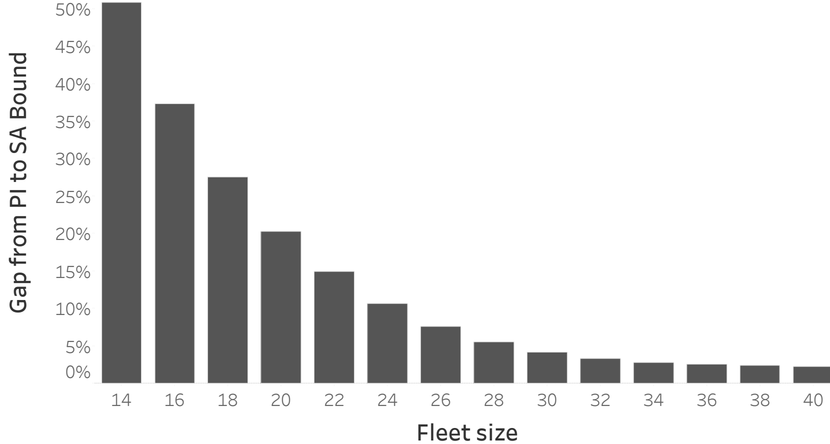

Figure 4: The mean percent difference of the SA bound relative to the PI bound over 15 episodes of an instance with 1400 requests for varying fleet sizes. Note: computed as $\frac{SA-PI}{PI}$, where SA and PI are (upper bounds on) the profit earned by a policy that serves all requests and an optimal policy with perfect information, respectively.

gap between the bounds) at a fleet size of 38. Because the SA bound does not include the travel costs incurred outside of serving requests, the gap between the SA and PI bounds will generally be greater than zero. We note that the PI values here reflect the mixed PI bound discussed in §5.2.3, implying that the gaps may actually be larger than these results suggest.

6.2.2 Primary 1400/14 Instance Set.

The Dart and Drafter *training curves*, showing their increasing objective performance over the 750 training episodes, are provided in Figure 5. Dashed horizontal lines in the figure indicate the mean objective performance of Reopt and Random, and thin vertical lines indicate the points at which the Dart and Drafter agents surpass the mean performance of the Reopt agent. We see that learning happens quickly for both agents, with Dart surpassing Reopt after 96 training episodes and Drafter doing so after only 41 episodes. Both agents' objective performance is stable after 250 episodes. In Figure 6 we compare agents' performance on the 30 evaluation episodes for which the PI and SA dual bounds are available. We find that the Drafter agent is the best performing with a mean daily profit of $10,898, although there remains a large gap between this policy and the PI dual bound, and an even larger gap between the PI and SA dual bounds (as expected from the experiments in §6.2.1). We note that the reported PI bound is a mixed bound (see §5.2.3), as we are not able to solve any of the 30 episodes to optimality. After Drafter, Dart is the second-best performing agent, earning a mean daily profit of $9,627. Dart's performance is not significantly different from Reopt, which has a mean profit of $9,067. On this set of 30 instances, Drafter outperforms Reopt by 20.2%.

6.2.3 Scalability Tests.

After training on the 1400/14 instances, we then apply Drafter directly to instances two, five and ten times larger. With the primary instances of size 1400/14 being the basis with an *Instance Scale* of 1, we refer to these larger instances as having Instance Scales of 2, 5, and 10, which respectively correspond

Figure 5: Training curves for the Dart and Drafter agents, showing two-week performance averages (the average objective over an episode and the previous 13 episodes). Dashed horizontal lines depict mean objective values for the Random and Reopt agents. Vertical lines indicate where Dart and Drafter surpass Reopt's mean objective achievement.

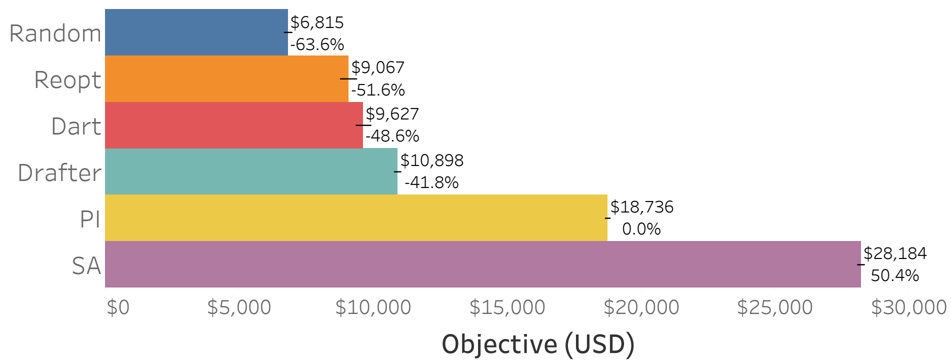

Figure 6: Agents' objective performance over 30 evaluation episodes of size 1400/14 for which dual bounds are available. Parenthesized values indicate the gap to the PI bound. Black marks indicate 95% confidence intervals.

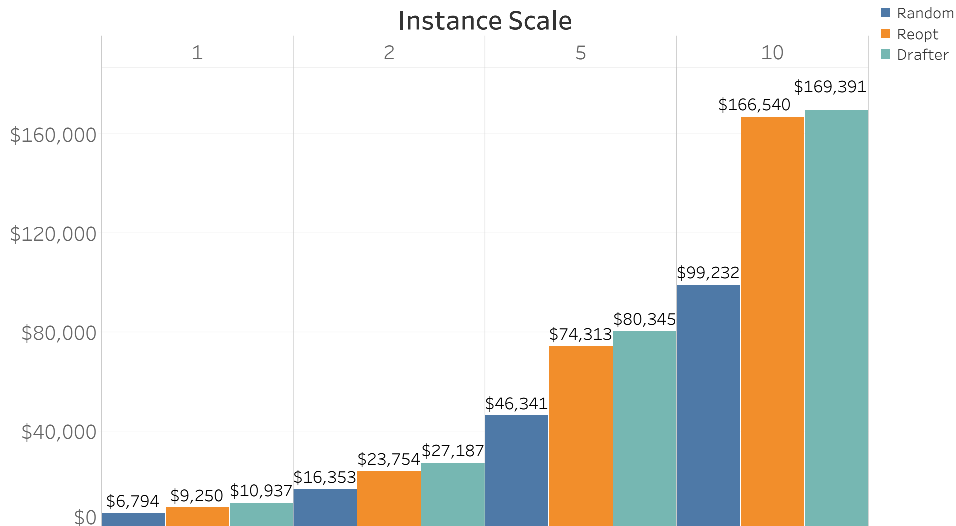

Figure 7: Agents' objective performance over 200 evaluation episodes of various instance scales.

to V/R ratios of 2800/28, 7000/70, and 14000/140 (all other instance parameters remain the same). Across all scales, Drafter is the best performing agent. Its advantage over Reopt is largest in the 1400/14 instances, with an objective 18.2% better on average over the 200 evaluation episodes. This edge over Reopt decreases with increasing instance scale, to 14.5% for the 2800/28 instances, 8.1% for the 7000/70 instances, and 1.7% in the 14000/140 instances. This relationship is shown in Figures 7 and 8.

We provide further comparison between the agents in Figure 9 which shows three metrics of interest for a fleet operator: as measures of customer service, we consider average customer waiting time and number of requests served; and as a measure of efficiency, we consider the proportion of *productive travel* (the percentage of the total distance traveled by the fleet with a customer on board). We find that Drafter serves the greatest percentage of customer requests across all instances sizes, serving 44% of requests in the 1400/14 instances (compared to 34% for Reopt), 54% in the 2800/28 instances (44% for Reopt), and 66% in the 14000/140 instances (61% for Reopt). Although serving fewer requests, Reopt does achieve the lowest waiting time for the customers that it does serve. This is true across all instance scales. Reopt serves customers 10 s faster than Drafter in the 1400/14 instances, 15.6 s faster in the 2800/28 instances, 23.9 s faster in the 7000/70 instances, and 27.2 s faster in the 14000/140 instances. Lastly, we observe that Reopt also has a greater proportion of productive travel than Drafter, achieving an average of 74% across instance scales, compared to an average of 50% for Drafter.

6.3 Discussion

While often in dynamic optimization no dual bound is available against which to compare policy performance, we offer two here. Although the gaps from these bounds to our agents are large, the results still serve to illustrate that even a loose bound has utility. For example, comparative analysis of the PI and SA bounds suggests that access to perfect information is of significant value in the E-RPC: for instances with a V/R ratio of 1400/38 and greater, the bounds are nearly equivalent, indicating that an optimal policy with PI is consistently able to serve all requests.

Our computational experiments primarily set out to compare the Reopt and Drafter agents. In the experiments posed here, Drafter is the apparent winner as it consistently achieves both the best objective

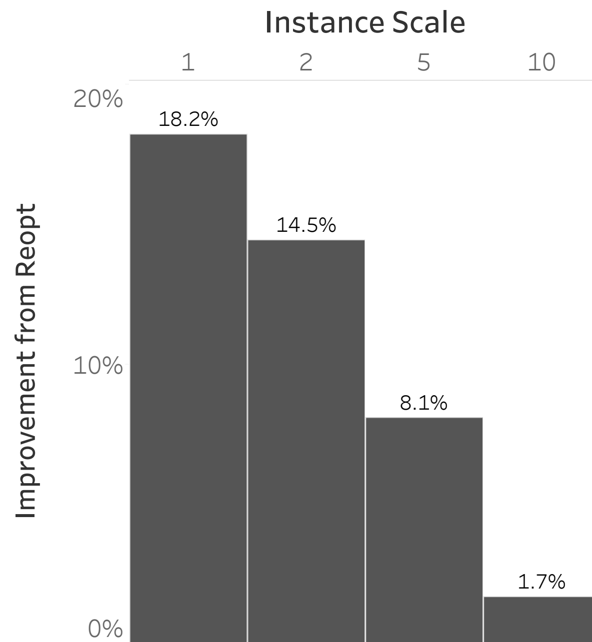

Figure 8: The improvement in objective offered by Drafter over Reopt over 200 evaluation episodes of instances of varying scales.

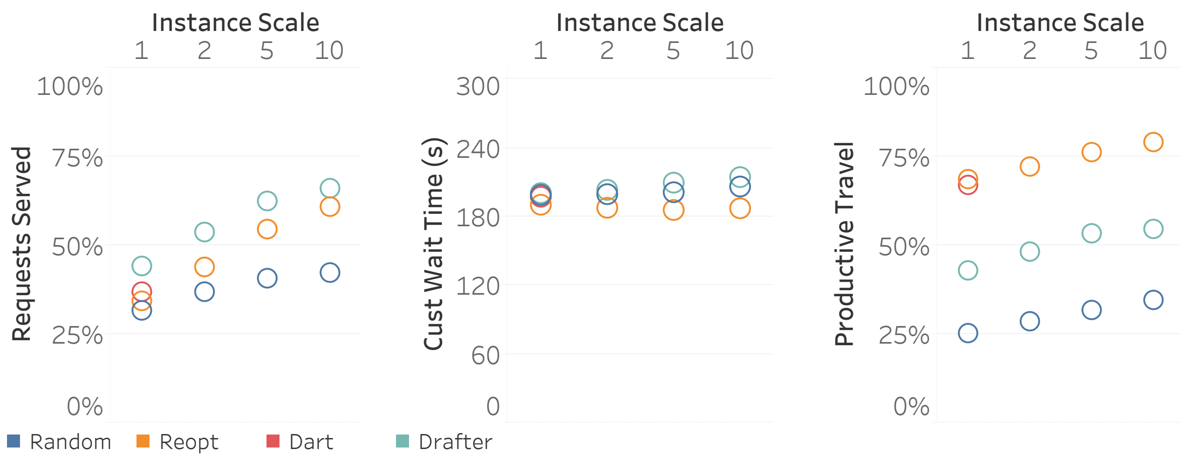

Figure 9: Over 200 evaluation episodes of various instance scales, agents' performance as measured by (Left) the percent of requests that were served, (Center) the average customer waiting time, and (Right) the proportion of travel that is productive. Note that because it does not scale directly, Dart appears only for Instance Scale 1.

values and the greatest percent of customer requests served. These are arguably the most important criteria to an operator of a ridehail fleet. That Drafter consistently serves more requests, when taken alongside its higher waiting times (207 s on average, compared to 187 s for Reopt), suggests that Drafter is able to position the fleet so as to better cover the demand. Notably, Drafter requires no prior knowledge or assumptions about the shape of this demand to learn to do so — in the language of reinforcement learning, it is *model-free*. Rather than relying on, e.g., historical data or domain knowledge regarding parameters describing the demand's spatial and temporal distributions, Drafter garners sufficient working knowledge of the demand during its training to determine which actions best allow it to serve future requests and ultimately achieve greater objective values. Being model-free also implies that Drafter is flexible to various relationships between actions and rewards, because this relationship is learned by the agent. That is, while we trained and evaluated Drafter on the reward function defined by equation (2), we could have equally used, e.g., a deterministic reward function or one which incorporated additional revenue or cost metrics. Although displaying inferior performance than Drafter and lacking its scalability, Dart is also model-free and therefore enjoys these same benefits. Drafter's scalability should be encouraging for operators of ridehail companies, as it suggests robustness to changes in problem instances. That is, in the event that demand is slow on a particular day or one or more vehicles are unavailable (e.g., due to repairs), Drafter should still provide reliable service.

We note the encroaching objective performance of Reopt with increasing Instance Scales relative to Drafter, with Drafter's advantage shrinking from 18.2% in the 1400/14 instances to 1.7% in the 14000/140 instances. This result is not surprising. First, Drafter is acting in an environment that is increasingly different from the one in which it was trained. Additionally, with more requests pooled in a single decision-making epoch, Reopt is armed with increasing information with which to make decisions. This ability to wait-and-see is an innate advantage possessed by reoptimization-style agents. Future research should investigate these two points in more detail. For example, for the former, while we lacked the computational resources to do so here, it would be interesting to compare the performance of a scaled Drafter agent (like the one used here) against a Drafter agent that was trained directly on instances of the larger size. This would answer the question as to how much of Drafter's relative decline was due to the difference in its evaluation and training environments, rather than Reopt simply performing better at scale. In the event that, under this comparison, the gap between Drafter and Reopt continues to be smaller for larger instances, then it would suggest that Reopt-style agents may simply have an advantage at larger instance scales, due to their ability to more completely observe demand before making decisions. Regardless, it would be a worthy endeavor to create an MDP problem model that would accommodate a deep-RL agent that could similarly pool requests for some amount of time before making a decision.

Lastly, we comment on the proportion of productive travel. This metric is primarily of interest to cities, as they aim to reduce congestion on roadways. Our results suggest that fleet operation costs alone may not be ample motivation for ridehail operators to minimize nonproductive travel: in the presence of realistic revenues and prices, Drafter found that it was still in its financial interest to pursue nonproductive vehicle movements (in order to serve future requests). To combat such behavior, cities may wish to implement congestion pricing (see, e.g., the review in Gu et al. 2018). Future research may investigate how learned agent behavior varies under different pricing schemes, as one would expect that nonproductive travel would decrease with increased congestion pricing.

7 Concluding Remarks

We considered the E-RPC, the problem faced by an operator of a ridehail service with a fleet composed of centrally-controlled electric vehicles. To solve the E-RPC, we developed two policies, Dart and Drafter, derived with deep reinforcement learning. To serve as a comparison, we consider a reoptimization policy from the ridehailing literature, as well as a randomly-acting policy that serves as a lower bound. We develop two dual bounds on the value of an optimal policy, including the value of an optimal policy with perfect information. To establish this bound, we decompose a relaxed version of the E-RPC into time- and energy-feasibility subproblems, and solve them via a Benders-like decomposition. We offer an analysis of when this more complex dual bound offers value beyond that of a simpler proposed dual bound.

We perform computational experiments on instances derived from real world data for ridehail operations in New York City in 2018. Across instances of varying scales, we consistently find the deep RL-based Drafter agent to be the best performing. On instance scales for which it was specifically trained, it achieved an objective 18.2% better than the reoptimization policy. We further show that Drafter is scalable, finding that, on instances two, five, and ten times larger than any on which it was trained, it outperforms other policies by 14.5%, 8.1%, and 1.7%, respectively. These results ultimately suggest there are opportunities for deep reinforcement learning in solving problems like the E-RPC. By more efficiently utilizing finite resources, these methods have the potential to promote greater sustainability in the domain of transportation and logistics.

Acknowledgements

Nicholas Kullman is grateful for support from the Société des Professeurs Français et Francophones d'Amérique. Justin Goodson wishes to express appreciation for the support from the Center for Supply Chain Excellence at the Richard A. Chaifetz School of Business. Martin Cousineau and Jorge Mendoza would like to thank HEC Montréal and the Institute for Data Valorization (IVADO) for funding that contributed to this project; Jorge Mendoza is additionally grateful for funding from HEC Montréal's Professorship on Clean Transportation Analytics. This research was also partly funded by the French Agence Nationale de la Recherche through project e-VRO (ANR-15-CE22-0005-01).

A Updating Vehicles' Job Descriptions

Upon choosing action $\mathbf{a}$ from state s, we immediately update vehicles' job descriptions. We describe this process here. For the following discussion, let the current location of vehicle v be (v_x, v_y).

NOOP.

Vehicles receiving the NOOP instruction ($a_v = \emptyset$) see no change to their existing job descriptions.

Repositioning/Recharging.

Let the location to which the vehicle is instructed to reposition be $a_v = c$ with location (c_x, c_y). We set the first job's type to repositioning ($j_m^{(1)} = 2$), its origin to the vehicle's current location ($j_o^{(1)} = (v_x, v_y)$), and its destination to c ($j_d^{(1)} = (c_x, c_y)$). We then set the second job type to charging ($j_m^{(2)} = 0$) which begins and ends at c ($j_o^{(2)} = j_d^{(2)} = (c_x, c_y)$). After recharging, the vehicle then idles at the station: $j_m^{(3)} = 0$ and $j_o^{(3)} = j_d^{(3)} = (c_x, c_y)$.

Serving Request.

Let the new request have origin (o_x, o_y) and destination (d_x, d_y).

No work-in-process. If the vehicle is not already serving a request ($j_m^{(1)} \neq 4$), then its first job is updated to type *preprocess* ($j_m^{(1)} = 3$) with origin $j_o^{(1)} = (v_x, v_y)$ and destination $j_d^{(1)} = (o_x, o_y)$. Its second job is then serving the customer, so we set the type to serve ($j_m^{(2)} = 4$), the origin $j_o^{(2)} = (o_x, o_y)$, and the destination $j_d^{(2)} = (d_x, d_y)$. The vehicle's third job is then empty, so we set $j_m^{(3)} = j_o^{(3)} = j_d^{(3)} = \emptyset$.

Work-in-process. If the vehicle is currently serving an existing customer ($j_m^{(1)} = 4$), then the first job is unchanged. The second job is *preprocess* ($j_m^{(2)} = 3$), as the vehicle moves from the drop-off location of its current customer ($j_o^{(2)} = j_d^{(1)}$) to the pick-up location of the new customer ($j_d^{(2)} = (o_x, o_y)$). The vehicle's third job is then to serve the customer request, so we set the type to serve ($j_m^{(3)} = 4$), the origin $j_o^{(3)} = (o_x, o_y)$, and the destination $j_d^{(2)} = (d_x, d_y)$.

B Agent Details: Hyperparameters & Attention Mechanism

We provide here additional details about the training and implementation of the Dart and Drafter agents. Hyperparameters, chosen based on empirical results, are given in Table 3. The rest of this section describes the attention mechanism used in Drafter's DQN.

The attention mechanism used by the Drafter agent is depicted in Figure 1. We begin by creating an *embedding* g_v of each vehicle's representation x_v using a single dense layer called the "vehicle embedder." An embedding of some input is simply an alternative representation of that input. For example, for a vehicle representation $x_v \in [0,1]^2 \times \{0,1\}^{|\mathcal{L}|}$ (see §4.1.3) the embedder maps it to $g_v \in \mathbb{R}^{128}$ via $f(W_a x_v + \mathbf{b})$, where W_a is a $128 \times (2 + |\mathcal{L}|)$ matrix of trainable weights, $\mathbf{b}$ is a 128-length vector of trainable weights, and f is the activation function (here a rectified linear unit, $ReLU$). Similar to vehicle embeddings g_v, we create an embedding $h \in \mathbb{R}^{64}$ of the request x_r using another single dense layer (the "request embedder"). We then concatenate each vehicle's embedding g_v with the request embedding h

Table 3: Agents' hyperparameters.

Parameter	Dart	Drafter
Optimizer (learning rate)	Adam (0.001)	Adam (0.001)
Loss function	Huber	Huber
Discount factor γ	0.999	0.999
Memory capacity	1,000,000	1,000,000
Steps prior to learning M_{start}	2,000	2,000
Training frequency M_{freq}	100	100
Batch size M_{batch}	32	32
Initial epsilon ϵ_i	1	1
Final epsilon ϵ_f	0.01	0.1
Epsilon decay steps ϵ_N	75,000	75,000
PER type	None (uniform)	proportional
PER α	-	0.6
PER β_0	-	0.4
PER beta decay steps	-	600,000
Target network update frequency M_{update}	5,000	5,000
n-step learning	3	3
DQN activation functions	ReLU	ReLU*
Number of hidden layers (pre-dueling, advantage stack, value stack)	(1,2,2)	(2,2,2)**

* *With the exception of the layers in the attention mechanism as described in §B.*

** *For the inner DQN (see Figure 1).*

to produce joint embeddings $j_v = g_v \oplus h$. The j_vs and g_vs are then used to perform attention over the vehicles, creating a *fleet context* vector $C^F \in \mathbb{R}^{128}$ via

$$C^F = \sum_{v \in \mathcal{V}} \alpha_v g_v, \tag{34}$$

where $\alpha_v = \sigma(\mathbf{w} \cdot \tanh(W \cdot j_v))$ is a scalar weighting vehicle v, σ is the sigmoid activation function, $\mathbf{w}$ is a trainable vector of weights, and W is a trainable matrix of weights (the summation in equation (34) is performed element-wise). This attention mechanism is similar to the one used to produce the fleet context in Holler et al. (2019), with the notable difference that here we also incorporate the request. As the request s_r is likely to influence which vehicles are relevant in a given state, its inclusion in the attention mechanism should yield a more descriptive fleet context C^F. The fleet context vector is then used in the production of Q-values for each vehicle as described in §4.1.4.

C Froger et al. (2019) FRVCP Labeling Algorithm

We provide here additional details on the labeling algorithm from Froger et al. (2019) and our modifications to accommodate time constraints on the customers. We will refer to the sequence of requests to be served by vehicle v, as determined by the master problem (§5.2), by $r_v = (r_v^1, r_v^2, \ldots)$.

C.1 Algorithm Overview

To find the optimal recharging instructions given a request sequence r_v, the FRVCP is reformulated as a resource-constrained shortest path problem. The algorithm then works by setting labels at nodes on a graph $\mathcal{G}'_v$ (see example in Figure 10) that reflects the vehicle's assigned request sequence r_v and possible charging station visits. Labels are defined by *state-of-charge* (SoC) *functions*. SoC functions are piecewise-linear functions comprised of *supporting points* $z = (z^t, z^q)$ that describe a state of departure from a node in $\mathcal{G}'_v$ in terms of time z^t and battery level z^q. See Figure 11 for an example.

During the algorithm's execution, labels are extended along nodes in $\mathcal{G}'_v$. When extending labels, SoC functions are shifted by the travel time and energy between nodes, and supporting points resulting in infeasible (negative) charges are pruned. When extending a label to a charging station node, we create new supporting points that correspond to the *breakpoints* in the charging curve at that station – energy levels at which the charging rate changes. (Note that here, we assume linear charging until the vehicle reaches full battery, at which point it begins idling. This breakpoint corresponds to z_2 in the left graph of Figure 11.) We continue to extend labels along nodes in $\mathcal{G}'_v$ until either the destination node $r_v^{|r_v|}$ is reached or there are no feasible label extensions. If the destination node is reached, the algorithm returns the earliest arrival time to that node based on the label's SoC function and the sequence is deemed to be energy feasible; if the destination node cannot be reached, the sequence is deemed energy-infeasible and the algorithm returns the first request node to which it could not extend a label. Bounds on energy and time are established in pre-processing and are used alongside dominance rules during the algorithm's execution in order to improve its efficiency. For complete details on the algorithm, we refer the reader to Froger et al. (2019).

C.2 Algorithm Modifications

We modify the algorithm to accommodate time constraints for the requests in r_v, which are not considered in the original implementation. We assume that the vehicle is eligible to depart its initial location, denoted

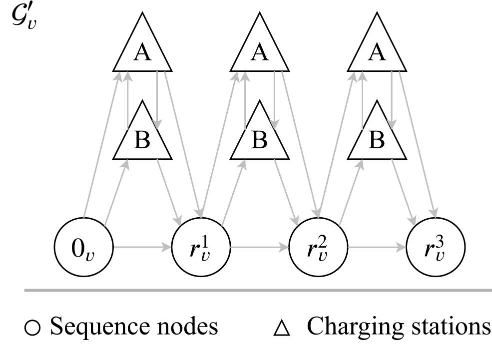

Figure 10: A depiction of the graph G'_v for a vehicle assigned to serve three customer requests. In the example, we assume there are two stations, A and B, at which the vehicle can recharge.

0_v, at time t_v^p with charge q_v^p. Moving between locations a and b requires energy $q_{a,b}^s$ and time $t_{a,b}^s$ (if b is a request, the values reflect the time and energy to reach its destination via its origin). For this discussion, additional information about the algorithm beyond the overview in §C.1 may be necessary, for which we refer the reader to the description of Algorithm 3 in §5.3 and Appendix E of Froger et al. (2019).

The first modification serves to include the option of idling at charging stations. We add a point in the charging function at (∞, Q) as shown in Figure 11. This allows the vehicle to idle at a CS after it has fully charged its battery. Second, when extending a label to a request, we prune supporting points at the request node based on the request's time constraints. Specifically, all supporting points that are too early are shifted later in time. This may result in multiple supporting points with the same time but different energy levels, in which case we keep only the supporting point with maximum energy. Supporting points whose times are too late are eliminated, and a new supporting point is established where the SoC function intersects the end of the customer's time window. These processes are shown in Figure 11. Finally, when extending a label from one request node directly to another, if a supporting point has been shifted later in time by some amount Δt (like z_1'' in Figure 11), then the point incurs a charge penalty Δq. This reflects the constraint that vehicles may not idle at request nodes, so we assume that the vehicle has continued driving (e.g., "circling the block") during the time Δt and has thus drained its battery by the amount Δq. Note that such penalties are not incurred when extending to or from charging station nodes in $\mathcal{G}'_v$ as these nodes have no time constraints.

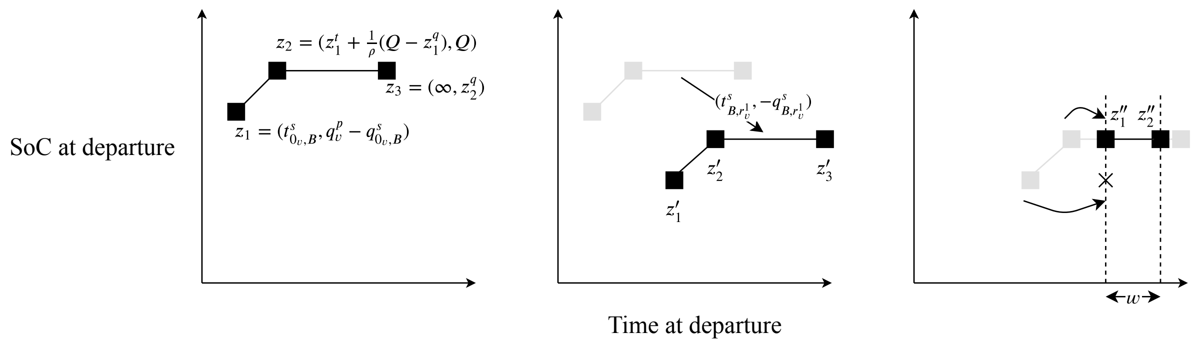

Figure 11: SoC functions for labels extended from 0_v to B to r_v^1. (Left) The SoC function at B after arriving directly from 0_v with supporting points corresponding to immediate departure without recharging (z_1), recharging completely (z_2), and subsequent indefinite idling (z_3). (Center) These points are then shifted by $(t^s_{B,r^1_v}, -q^s_{B,r^1_v})$ as we extend a label to r_v^1, resulting in points z_1', z_2', z_3'. (Right) Supporting points must lie within customer r_v^1's time window w^*. Point z_1' is eliminated since it is dominated after this shift (indicated by the "x"), having less charge than the new point z_1''. Point z_3' is eliminated and the new point z_2'' is created where the SoC function intersects the end of the time window. Note that there is no charge penalty incurred here, as the label is being extended from a charging station.

References

Lina Al-Kanj, Juliana Nascimento, and Warren B Powell. Approximate dynamic programming for planning a ride-sharing system using autonomous fleets of electric vehicles. *arXiv preprint arXiv:1810.08124*, 2018.

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

Omid Bahrami, Jim Gawron, Jack Kramer, and Bob Kraynak. Flexbus: Improving public transit with ride-hailing technology, December 2017. `http://sustainability.umich.edu/media/files/dow/Dow-Masters-Report-FlexBus.pdf`, Accessed December 2019.

Dimitris Bertsimas, Patrick Jaillet, and Sébastien Martin. Online vehicle routing: The edge of optimization in large-scale applications. *Operations Research*, 67(1):143–162, 2019.

Joschka Bischoff and Michal Maciejewski. Simulation of city-wide replacement of private cars with autonomous taxis in berlin. *Procedia Computer Science*, 83:237–244, 2016.

Patrick M Bösch, Felix Becker, Henrik Becker, and Kay W Axhausen. Cost-based analysis of autonomous mobility services. *Transport Policy*, 64:76–91, 2018.

Anton Braverman, Jim G Dai, Xin Liu, and Lei Ying. Empty-car routing in ridesharing systems. *Operations Research*, 67(5):1437–1452, 2019.

T Donna Chen, Kara M Kockelman, and Josiah P Hanna. Operations of a shared, autonomous, electric vehicle fleet: Implications of vehicle & charging infrastructure decisions. *Transportation Research Part A: Policy and Practice*, 94:243–254, 2016.

Geoff Clarke and John W Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

Florian Dandl, Michael Hyland, Klaus Bogenberger, and Hani S Mahmassani. Evaluating the impact of spatio-temporal demand forecast aggregation on the operational performance of shared autonomous mobility fleets. *Transportation*, 46(6):1975–1996, 2019.

Edison Electric Institute. Electric vehicle sales: Facts and figures, October 2019. `https://www.eei.org/issuesandpolicy/electrictransportation/Documents/FINAL_EV_Sales_Update_Oct2019.pdf`, Accessed January 2020.

Daniel J Fagnant and Kara Kockelman. Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice*, 77:167–181, 2015.

Daniel J Fagnant and Kara M Kockelman. The travel and environmental implications of shared autonomous vehicles, using agent-based model scenarios. *Transportation Research Part C: Emerging Technologies*, 40: 1–13, 2014.

Aurélien Froger, Jorge E Mendoza, Ola Jabali, and Gilbert Laporte. Improved formulations and algorithmic components for the electric vehicle routing problem with nonlinear charging functions. *Computers & Operations Research*, 104:256–294, 2019.

Michel Gendreau, Gilbert Laporte, and Jean-Yves Potvin. Metaheuristics for the capacitated VRP. In Paolo Toth and Daniele Vigo, editors, *The vehicle routing problem*, chapter 6, pages 129–154. SIAM, 2002.

Ziyuan Gu, Zhiyuan Liu, Qixiu Cheng, and Meead Saberi. Congestion pricing practices and public acceptance: A review of evidence. *Case Studies on Transport Policy*, 6(1):94–101, 2018.

H.V. Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 2094–2100. AAAI Press, 2016.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *32nd AAAI Conference on Artificial Intelligence*, pages 3207–3214. AAAI Press, 2018.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *32nd AAAI Conference on Artificial Intelligence*, pages 3215–3222. AAAI Press, 2018.

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

John Holler, Risto Vuorio, Zhiwei Qin, Xiaocheng Tang, Yan Jiao, Tiancheng Jin, Satinder Singh, Chenxi Wang, and Jieping Ye. Deep reinforcement learning for multi-driver vehicle dispatching and repositioning problem. *arXiv preprint arXiv:1911.11260*, 2019.

Sebastian Hörl, Claudio Ruch, Felix Becker, Emilio Frazzoli, and Kay W Axhausen. Fleet operational policies for automated mobility: A simulation assessment for zurich. *Transportation Research Part C: Emerging Technologies*, 102:20–31, 2019.

Michael Hyland and Hani S Mahmassani. Dynamic autonomous vehicle fleet operations: Optimization-based strategies to assign avs to immediate traveler demand requests. *Transportation Research Part C: Emerging Technologies*, 92:278–297, 2018.

Riccardo Iacobucci, Benjamin McLellan, and Tetsuo Tezuka. Optimization of shared autonomous electric vehicles operations with charge scheduling and vehicle-to-grid. *Transportation Research Part C: Emerging Technologies*, 100:34–52, 2019.

Erick C Jones and Benjamin D Leibowicz. Contributions of shared autonomous vehicles to climate change mitigation. *Transportation Research Part D: Transport and Environment*, 72:279–298, 2019.

Jaeyoung Jung, R Jayakrishnan, and Keechoo Choi. Shared-taxi operations with electric vehicles. *Institute of Transportation Studies Working Paper Series, Irvine, Calif*, 2012.

Namwoo Kang, Fred M Feinberg, and Panos Y Papalambros. Autonomous electric vehicle sharing system design. *Journal of Mechanical Design*, 139(1):011402, 2017.

Andrej Karpathy. Deep reinforcement learning: Pong from pixels, May 2016. `http://karpathy.github.io/2016/05/31/rl/`, Accessed January 2020.

Nicholas Kullman, Aurelien Froger, Jorge E. Mendoza, and Justin C. Goodson. frvcpy: An open-source solver for the fixed route vehicle charging problem. *INFORMS Journal on Computing*, 2020. In press.

Charly Robinson La Rocca and Jean-François Cordeau. Heuristics for electric taxi fleet management at teo taxi. *INFOR: Information Systems and Operational Research*, 0(0):1–25, 2019.

Der-Horng Lee, Hao Wang, Ruey Long Cheu, and Siew Hoon Teo. Taxi dispatch system based on current demands and real-time traffic conditions. *Transportation Research Record*, 1882(1):193–200, 2004.

Minne Li, Zhiwei Qin, Yan Jiao, Yaodong Yang, Jun Wang, Chenxi Wang, Guobin Wu, and Jieping Ye. Efficient ridesharing order dispatching with mean field multi-agent reinforcement learning. In *The World Wide Web Conference*, pages 983–994. ACM, 2019.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

Lyft. Why 130,000 lyft passengers were ready to ditch their personal cars in less than 24 hours, December 2018. `https://blog.lyft.com/posts/ditchyourcardata`, Accessed December 2019.

Fei Miao, Shuo Han, Shan Lin, John A Stankovic, Desheng Zhang, Sirajum Munir, Hua Huang, Tian He, and George J Pappas. Taxi dispatch with real-time sensing data in metropolitan areas: A receding horizon control approach. *IEEE Transactions on Automation Science and Engineering*, 13(2):463–478, 2016.

Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent models of visual attention. In *Proceedings of the 27th International Conference on Neural Information Processing Systems-Volume 2*, pages 2204–2212. MIT Press, 2014.

Alejandro Montoya, Christelle Guéret, Jorge E Mendoza, and Juan G Villegas. The electric vehicle routing problem with nonlinear charging function. *Transportation Research Part B: Methodological*, 103:87–110, 2017.

National Renewable Energy Laboratory. Alternative fuels data center, 2019. Data retrieved 09/2019 from `https://www.nyserda.ny.gov/All-Programs/Programs/Drive-Clean-Rebate/Charging-Options/Electric-Vehicle-Station-Locator`.

New York City Taxi & Limousine Commission. TLC trip record data, 2018. Data retrieved 09/2019 from `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`.

NYC OpenData. NYC taxi zones, 2019. Data retrieved 09/2019 from `https://data.cityofnewyork.us/Transportation/NYC-Taxi-Zones/d3c5-ddgc`.

Takuma Oda and Carlee Joe-Wong. Movi: A model-free approach to dynamic fleet management. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2708–2716. IEEE, 2018.

Takuma Oda and Yulia Tachibana. Distributed fleet control with maximum entropy deep reinforcement learning. In *NeurIPS 2018 Machine Learning for Intelligent Transportation Systems Workshop*, 2018.

Office of Transportation and Air Quality. U.S. Transportation Sector Greenhouse Gas Emissions 1990-2017, June 2019. URL `https://nepis.epa.gov/Exe/ZyPDF.cgi?Dockey=P100WUHR.pdf`. EPA-420-F-19-047.

Samuel Pelletier, Ola Jabali, and Gilbert Laporte. 50th anniversary invited article – goods distribution with electric vehicles: Review and research perspectives. *Transportation Science*, 50(1):3–22, 2016.

Jacob F Pettit, Ruben Glatt, Jonathan R Donadee, and Brenden K Petersen. Increasing performance of electric vehicles in ride-hailing services using deep reinforcement learning. *arXiv preprint arXiv:1912.03408*, 2019.

Harilaos N Psaraftis, Min Wen, and Christos A Kontovas. Dynamic vehicle routing problems: Three decades and counting. *Networks*, 67(1):3–31, 2016.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In *4th International Conference on Learning Representations*, 2016.

Todd W. Schneider. Taxi and ridehailing usage in new york city, 2019. `https://toddwschneider.com/dashboards/nyc-taxi-ridehailing-uber-lyft-data/`, Accessed September 2019.

Kiam Tian Seow, Nam Hai Dang, and Der-Horng Lee. A collaborative multiagent taxi-dispatch system. *IEEE Transactions on Automation Science and Engineering*, 7(3):607–616, 2009.

Jie Shi, Yuanqi Gao, Wei Wang, Nanpeng Yu, and Petros A Ioannou. Operating electric vehicle fleet for ride-hailing services with reinforcement learning. *IEEE Transactions on Intelligent Transportation Systems*, 2019.

Ashutosh Singh, Abubakr Alabbasi, and Vaneet Aggarwal. A distributed model-free algorithm for multi-hop ride-sharing using deep reinforcement learning. In *NeurIPS 2019 Machine Learning for Autonomous Driving Workshop*, 2019.

Peter Slowik, Lina Fedirko, and Nic Lutsey. Assessing ride-hailing company commitments to electrification, February 2019. `https://theicct.org/sites/default/files/publications/EV_Ridehailing_Commitment_20190220.pdf`, Accessed January 2020.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018. ISBN 9780262352703.

Tesla. Supercharging cities, 2017. `https://www.tesla.com/blog/supercharging-cities`, Accessed December 2019.

Z. Wang, T. Schaul, M. Hessel, H.V. Hasselt, M. Lanctot, and N.D. Freitas. Dueling network architectures for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of*

The 33rd International Conference on Machine Learning, volume 48, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR.

Zhe Xu, Zhixin Li, Qingwen Guan, Dingshui Zhang, Qiang Li, Junxiao Nan, Chunyang Liu, Wei Bian, and Jieping 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*, pages 905–913. ACM, 2018.

Lingyu Zhang, Tao Hu, Yue Min, Guobin Wu, Junying Zhang, Pengcheng Feng, Pinghua Gong, and Jieping Ye. A taxi order dispatch model based on combinatorial optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2151–2159. ACM, 2017.

Rick Zhang and Marco Pavone. Control of robotic mobility-on-demand systems: A queueing-theoretical perspective. *The International Journal of Robotics Research*, 35(1-3):186–203, 2016.