Manuscript File Click here to view linked References

Deep Reinforcement Learning for Crowdsourced Urban Delivery

Tanvir Ahamed[1], Bo Zou[1*], Nahid Parvez Farazi[1], Theja Tulabandhula[2]

[1] Department of Civil, Materials, and Environmental Engineering, University of Illinois at Chicago

[2] Department of Information and Decision Sciences, University of Illinois at Chicago

Abstract: This paper investigates the problem of assigning shipping requests to *ad hoc* couriers in the context of crowdsourced urban delivery. The shipping requests are spatially distributed each with a limited time window between the earliest time for pickup and latest time for delivery. The *ad hoc* couriers, termed crowdsourcees, also have limited time availability and carrying capacity. We propose a new deep reinforcement learning (DRL)-based approach to tackling this assignment problem. A deep Q network (DQN) algorithm is trained which entails two salient features of experience replay and target network that enhance the efficiency, convergence, and stability of DRL training. More importantly, this paper makes three methodological contributions: 1) presenting a comprehensive and novel characterization of crowdshipping system states that encompasses spatial-temporal and capacity information of crowdsourcees and requests; 2) embedding heuristics that leverage information offered by the state representation and are based on intuitive reasonings to guide specific actions to take, to preserve tractability and enhance efficiency of training; and 3) integrating rule-interposing to prevent repeated visiting of the same routes and node sequences during routing improvement, thereby further enhancing the training efficiency by accelerating learning. The computational complexities of the heuristics and the overall DQN training are investigated. The effectiveness of the proposed approach is demonstrated through extensive numerical analysis. The results show the benefits brought by the heuristics-guided action choice, rule-interposing, and having time-related information in the state space in DRL training, the near-optimality of the solutions obtained, and the superiority of the proposed approach over existing methods in terms of solution quality, computation time, and scalability.

Keywords: Crowdshipping, deep reinforcement learning, deep Q network, pickup and delivery, state representation, heuristics-guided action choice, rule-interposing.

* Corresponding author. Email: bzou@uic.edu.

1 Introduction

This paper investigates a static crowdshipping problem with spatially distributed request pickup and delivery locations, using "crowdsourcees" who are ordinary people and also spatially distributed, and have some available time to perform delivery for income earning. A delivery service provider (DSP) centrally assigns requests to crowdsourcees to minimize total shipping cost (TSC). Distributed locations of requests are common for pickup-delivery from restaurants, grocery stores, and retail shops to customers, and even for document delivery between different office locations. We consider that each request has a narrow time window (e.g., two hours) between earliest pickup and latest delivery. In addition, crowdsourcees inform the DSP of their available time. Each crowdsourcee has a limited carrying capacity. Thus, the assignment needs to respect pickup-delivery time windows of requests, and time availability and carrying capacity of crowdsourcees. In shipping cost calculation, we consider that a crowdsourcee is paid a fixed rate ($/minute) when carrying a request. If a request is not assigned to a crowdsourcee, the request will be picked up and delivered by a backup vehicle, which is more expensive.

Following the above description, the crowdshipping problem can be viewed as a specific type of pickup-and-delivery problem and belongs to the broad category of vehicle routing problems (VRP). While many integer programming models and heuristic algorithms have been developed for solving similar problems, the novelty of this paper is that we propose, for the first time in the literature, an approach that leverages deep reinforcement learning (DRL)—more specifically deep Q learning (DQN)—to frame and solve the constrained crowdsourcee-shipping request assignment problem. Two salient features of DQN are experience replay and target network which can enhance efficiency, convergence, and stability in DRL training. Our work goes beyond simple adoption of the DQN algorithm in the existing literature, by making three major methodological contributions as follows.

The first contribution is on a novel representation of system states for the crowdshipping problem. Due to the combinatorial nature of the crowdshipping problem and the heterogeneity of both requests and crowdsourcees in terms of time and carrying capacity, the states of a crowdshipping system cannot be represented by one or a few metrics. A comprehensive representation must in some way capture the sequence of pickup and delivery nodes on each crowdsourcee route. A node corresponds to a physical location (with longitude and latitude information), which can be the origin of a crowdsourcee, the pickup location of a request, or the delivery location of a request. Yet routing sequence alone is not enough to reflect the fact that both requests and crowdsourcees are time sensitive: on the one hand, each request has a limited time window between the earliest possible pickup and the latest delivery (e.g., 2 hours).

On the other hand, by dedicating one's time to crowdshipping, a crowdsourcee also has limited time availability. The time information about requests and crowdsourcees, which changes as crowdsourcee routes are constantly created and improved, is an inherent part of the system state that helps the DRL agent make informed routing decisions, especially with respect to what requests need be considered first and what crowdsourcee routes may be given higher priority given time availability and delivery urgency. To this end, a novel representation of system states that leverages the notion of information array is proposed which encompasses not only static location information of request pickup and delivery nodes but information on crowdsourcee routing sequences, request-specific time availability, and crowdsourcee-specific time and capacity availability.

The second contribution is on embedment of heuristics-guided action choice in DRL. The combinatorial nature of the problem means that a very large number of different actions can be taken to construct and improve crowdsourcee routing. But enumerating all possible actions would be neither efficient nor practical in DRL training. To preserve training tractability, we abstract the action space into five general types of actions for assigning or improving the assignment of requests to crowdsourcees: 1) inserting an unassigned request to a crowdsourcee route (insertion); 2) moving an assigned request to another place in the same crowdsourcee route (intra-route move); 3) moving an assigned request to a different crowdsourcee route (inter-route move); 4) exchanging the positions of two requests that are assigned to two different crowdsource routes (1-exchange); 5) do-nothing. As many possibilities for taking a specific action still exist given an action type, heuristics that leverage the information offered by our proposed state representation and are based on intuitive reasonings are designed to guide the specific action to take. Thus, each time when an action needs to be taken, we first employ the DQN algorithm to identify the action type. Then, the specific action given the action type is executed using the corresponding heuristic. We show that the embedment of heuristics-guided action choice significantly enhances DRL training efficiency and solution quality.

The third contribution is on integration of rule-interposing into DRL training and implementation. The rules aim to prevent certain routes or node sequences from being visited repeatedly during neighborhood moves (i.e., intra-route move, inter-route move, and 1-exchange) within a period of time, as repeated visiting discourages exploring more actions and may get the routing sequence trapped in local optimum, thus compromising the efficiency of DRL training. Specifically, we employ two rules that: 1) set up and update a priority list of crowdsourcee routes for each neighborhood move, based on criteria in line with the nature of the neighborhood moves. A crowdsourcee route that is chosen for a neighborhood move will be removed from the priority list and not considered for some period of time; 2) introduce Tabu tenure for the relative positions of pickup and delivery nodes. Two nodes that were

neighbored and are moved away are prohibited to be neighbored again for some period of time. With the two rules, computation efforts involved in repeatedly visiting routes or node sequences during neighborhood moves are spared, thereby enhancing the training efficiency by accelerating learning.

With the above three methodological contributions, the effectiveness of the proposed DRL-based approach to solve the crowdshipping problems of our interest is demonstrated through extensive numerical analysis. Our results show superiority of the trained DQN algorithm over existing methods in solution quality, computation time, and scalability. In addition, the obtained solutions are reasonably close to global optimum. Given that the training of DRL will be performed offline and a trained DRL model can solve problems in a matter of seconds, the proposed approach has significant potential for practical crowdshipping operations. Moreover, the proposed methodological framework, which in this paper tackles a more complicated type of pickup and delivery problems with time constraints from both "vehicles" (crowdsourcees) and "customers" (shipping requests), has the potential to be adapted to solving similar types of routing-related problems.

The remainder of the paper is structured as follows. Section 2 reviews and synthesizes the relevant literature. Section 3 provides a detailed presentation of the methodology including the fundamentals of reinforcement learning (RL) and DRL; information array, representation of states, actions, and rewards; the DQN algorithm for crowdshipping; and rule-interposing design. Section 4 implements the DRL model and discusses the results from extensive numerical experiments. Summaries and suggestions for future research are given in Section 5.

2 Literature review

Crowdshipping has garnered growing research attention in recent years (e.g., Wang et al., 2016; Kafle et al., 2017; Le et al., 2019; Arslan et al., 2020). However, DRL has not been considered as a way to guide request-crowdsourcee assignment. Given the focus of the paper on the methodological aspects of DRL for crowdshipping and the relevance of our problem to other types of freight delivery and passenger transportation problems that involve routing, in this section we review recent advances of DRL in solving related problems. We will synthesize the problem characteristics and DRL specifications in representative studies, based on which the uniqueness of our paper is then highlighted.

A basic version of routing problems is the traveling salesman problem concerning routing of a single vehicle. Bello et al. (2016) probably make one of the first attempts to combine reinforcement learning with neural networks to tackle traveling salesman problems. A pointer network comprising two recurrent neural networks for encoding and decoding and an attention function is trained with policy gradient. Kool et al. (2018) build on Bello et al.'s work and train an attention-based encoder-

decoder DRL model. Dai et al. (2017) use a graph embedding network to represent the policy to capture the property of a node in the context of its graph neighborhood. A fitted Q-learning is adopted to learn a greedy policy that is parameterized by the graph embedding network. For the TSP problems considered above, only spatial information of nodes is involved. Actions in DRL pertain to adding nodes—one at a time—to progressively construct the vehicle route.

The complexity of routing problems is augmented when extended to multiple routes, with time constraints, and with pickups and deliveries. For freight delivery problems, Nazari et al. (2018) consider a parameterized stochastic policy to solve VRP with limited vehicle capacity. The authors apply a policy gradient algorithm to optimize parameters of a stochastic policy. Chen et al. (2019) use multi-agent RL to train a courier dispatch policy to deal with goods pickups with time windows. To maintain the state-action space, RL is decentralized with each courier modeled as an agent. However, a decentralized approach may compromise modeling of courier coordination in undertaking pickup tasks. The problem considered in Yu et al. (2019), which deals with pickup and delivery with vehicle capacity constraints and delivery deadline, is more similar to our paper. Like Chen et al. (2019), the authors opt for a distributed neural optimization strategy with a pointer network and graph embedding to progressively develop a complete tour of each vehicle. More recently, Duan et al. (2020) propose a joint learning approach based on graph convolutional network with node feature (coordinates and demand) and edge feature (distance) as inputs, to solve capacitated VRP.

On the passenger side, the interest in adopting DRL for VRP arises with the proliferation of ridesharing. Oda and Joe-Wong (2018) propose a DQN-based framework that learns which zone an idle vehicle should go to. The learning is independent for each vehicle, which is assumed to have at most one rider onboard at any point in time. Singh et al. (2019) relax the assumption by allowing more than one rider in a ridesharing vehicle. However, the training remains decentralized, i.e., each vehicle solves its DQN problem without coordination with other vehicles in vicinity. In addition, it is possible in the study that a rider transfers from one vehicle to another, which is undesirable and not common in practice. Another distributed model-free algorithm using DQN to learn dispatch policies for each vehicle individually is developed by Al-Abbasi et al. (2019), in which training of a vehicle's dispatching policy again does not consider coordination with other vehicles.

As shown in Table 1a, most of the multi-vehicle routing problems in the DRL literature are different from the crowdshipping problem in this paper. Only Yu et al. (2019) on the freight side and Al-Abbasi et al. (2019) on the passenger side consider pickup and delivery with the possibility of a vehicle carrying multiple customers at the same time and without transfer. While vehicle capacity limit is accounted for in some papers, customer time window constraints are mostly not, only in Chen et al.

(2019) and Yu et al. (2019). Yet none takes into account limited time availability of vehicles, which is an essential characteristic in our crowdshipping problem (where crowdsourcees are "vehicles"). Except for Nazari et al. (2019) and Duan et al. (2020), all other works train each vehicle individually, probably due to the substantially augmented action space and consequently complexity of DQN training if all vehicles are considered together (note that in Duan et al. (2020), vehicle routes are constructed one at a time sequentially rather than simultaneously). However, centralized DQN would be more appropriate as a DSP has full control in request-crowdsourcee assignment.

Because of the richer features and centralized nature for crowdshipping, fully capturing the states of a crowdshipping system requires more involved and elaborate representation. As shown in Table 1b, the existing studies mostly have vehicle and/or customer locations in state representation, with limited consideration of time-related information for vehicles and customers. On the other hand, given that both crowdsourcees and requests have limited time windows and that heuristics-guided action choice embedded in our proposed DRL requires time-related information to proceed, incorporation of time-related information is critical. Furthermore, for performing the heuristics, information on routing sequence is needed, which is not included explicitly in any prior studies reviewed. Also related to the heuristics-embedding feature, the specification of action space in our work is richer than in the existing literature. Finally, no existing papers consider rule-interposing.

Table 1a: VRP characteristics considered in selected DRL studies and the present paper

		Problem characteristics				
		Pickup and delivery	**Consider "vehicle" capacity constraint**	**Considers limited time of "customers"**	**Considers limited time of "vehicles"**	**Centralized**
passenger	Oda and Joe-Wong (2018)	Yes, but one rider in a vehicle at a time	No	No	No	No
	Singh et al. (2019)	Yes, but a rider may transfer between vehicles in a trip	Yes	No	No	No
	Al-Abbasi et al. (2019)	Yes (ridesharing)	Yes	No	No	No
Freight	Nazari et al. (2018)	No	Yes	No	No	Yes (but only 1 vehicle in numerical analysis)
	Chen et al. (2019)	No (pickup only)	No	Yes	No	No
	Yu et al. (2019)	Yes	Yes	Yes	No	No
	Duan et al. (2020)	No	Yes	No	No	Yes (but vehicle routes are constructed one at a time)
	This paper	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Note: The term "vehicle" is quoted because in crowdshipping, "vehicles" would refer to crowdsourcees. Similarly, the term "customers" is quoted as "customers" would refer to shipping requests on the freight side.

178 **Table 1b:** DRL specifications in solving VRP in selected studies and the present paper

		State representation	Action characterization	Rule-interposing
Passenger	Oda and Joe-Wong (2018)	1. Vehicle location 2. Occupied/idle status 3. Destination of the vehicle 4. Number of available vehicles in each zone 5. Future demand of each zone	Which zone for an idle vehicle under study to go to	No
	Singh et al. (2019)	1. Vehicle location (in which zone) 2. Available seats of each vehicle 3. Rider pickup time 4. Rider destination 5. Number of vehicles in each zone 6. Predicted future rider demand	Which zone to which vehicles are dispatched	No
	Al-Abbasi et al. (2019)	1. Vehicle location 2. Number of available seats 3. Rider pickup time 4. Rider destination 5. When an occupied vehicle becomes available 6. Future rider demand	1. Whether the vehicle under study should pick up new riders 2. If yes, which zone to go to	No
Freight	Nazari et al. (2018)	1. Customer location 2. Customer demand	Which node to visit by a vehicle	No
	Chen et al. (2019)	1. Number of couriers and requests in each grid 2. Total price of requests in each grid 3. Distance between neighboring grids 4. Score (percent of fulfilled price in total price)	1. Target grid 2. Maximum patrol time in the grid	No
	Yu et al. (2019)	1. Available requests 2. Renewable energy generation points 3. Next stops of other vehicles in the system 4. Battery charging demand of each vehicle	What is the next stop in the tour of the vehicle	No
	Duan et al. (2020)	1. Coordinates and demand at each customer node 2. Adjacency among customer nodes 3. Distance between any two customer nodes	What is the next node to visit	No
	This paper	1. Crowdsourcee starting locations 2. Request pickup and delivery locations 3. Node precedence relation of crowdsourcee routes 4. Request slack time, unused service time, and occupation time 5. Crowdsourcee routing duration and remaining available time 6. Time and capacity violation of crowdsourcee routes	1. Inserting a request to a route 2. Intra-route move of a request 3. Inter-route move of a request 4. 1-exchange move of two requests in two routes 5. No action	Yes

179

3 Methodology

This section describes the crowdshipping-adapted DRL methodology. First, we introduce the fundamental ideas of RL and DRL. Then, we discuss how states, actions, and rewards which are essential elements of DRL are specified in crowdshipping. Building on the specifications, we detail the training process using DQN. Two key ideas are worth mentioning. First, DQN learns from how a policy—a decision rule which directs what type of action to take given a state—performed on previous instances and improves the policy over time. Knowing the action type, the specific action will be determined by a corresponding heuristic that leverages time-related information from the state space. By letting DQN focus on only a small set of abstracted action types, the heuristics-guided action choice preserves training tractability and consequently contributes to the scalability of the proposed approach. Second, solutions to a crowdshipping problem instance can be constructed progressively, one step at a time, which is amenable to the DRL framework.

3.1 Fundamental idea

RL is one of the three categories of machine learning (the other two are supervised learning and unsupervised learning) (Sutton and Barto, 2018). The tenet of RL is to train an agent such that the agent can optimize its behavior by accumulating and learning from its experiences of interacting with the environment. The optimality is measured as maximizing the total reward by taking consecutive actions. At each decision point, the agent has information about the current state of the environment and selects the best action based on his current experiences. The action taken transitions the environment to a new state. The agent gets some reward, i.e., reinforcement, as a signal of how good or bad the action taken is.

To formulate the decision process, RL employs MDP as the mathematical foundation to keep track of the progression of the decision process. To do so, the following notations are introduced. S is the set of states of the environment. A is the set of actions the agent can take. R is the set of possible rewards as a result of the agent taking an action at a given state. To illustrate, the environment is in state $s_t \in S$ at time step t. The agent takes an action $a_t \in A$. The action transitions the environment to a new state $s_{t+1} \in S$ at the next time step $t + 1$. Meanwhile, the agent receives a reward $r_t \in R$. The reward is a function of state-action pair: $r_t(s_t, a_t)$ (Fig. 1).

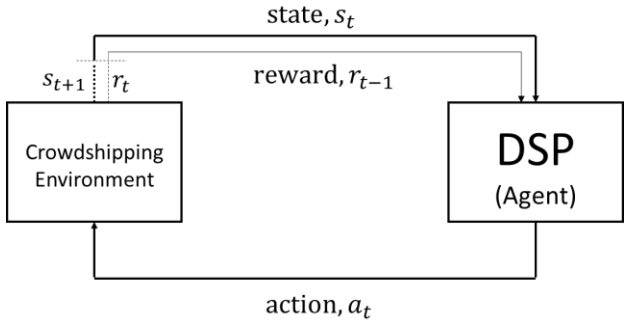


Fig. 1. Illustration of states, actions, and rewards

Since the actions are taken consecutively, the objective of the agent at any time step t is to maximize the cumulative reward, i.e., the return G_t, from t till the last time step T:

$$G_t = r_t + r_{t+1} + \cdots + r_T. \tag{1}$$

If we consider that the reward is received over a long period, a discount factor $\gamma \in [0,1]$ is often used to reflect discounting:

$$G_t = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t} r_T \tag{2}$$

In RL, a policy π is a mapping from states to probabilities of selecting each possible action. A value function V_π expresses the expected return when starting in state s and following policy π thereafter. At time step t, the value function can be written as:

$$V_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t} \gamma^k r_{t+k} \middle| s_t = s\right]. \tag{3}$$

Related to the value function, we define the value of taking action a in state s and following policy π thereafter, denoted as $Q_\pi(s,a)$. $Q_\pi(s,a)$ is termed action-value function, or "Q-function" of the state-action pair (s,a). The letter "Q" represents the quality of this state-action pair:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t} \gamma^k r_{t+k} \middle| s_t = s, a_t = a\right]. \tag{4}$$

It is desired to seek an optimal policy π^* such that $V_{\pi^*}(s) = \max_a Q_*(s,a)$, where $Q_*(s,a)$ means that the agent takes action a at state s and follows policy π^* thereafter. Clearly, if $Q_*(s,a)$ is known for every state-action pair (s,a), then π^* is also known. The problem of finding the optimal policy then becomes finding optimal Q-values $Q_*(s,a), \forall (s,a) \in S \times A$. To do so, one of the prominent algorithms is Q-learning (Watkins and Dayan, 1992). At a time step, the Q-function value (thereafter simplified as "Q-value") for a given state-action pair is updated using the following rule which is based on the Bellman optimality equation:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[r(s,a) + \gamma \max_{a' \in A} Q(s',a')\right] \quad (5)$$

where s' is the transitioned state after taking action a at state s. $r(s,a)$ is the associated reward. On the left-hand side of Eq. (5) is the updated $Q(s,a)$ value. On the right-hand side (RHS), $Q(s,a)$ and $Q(s',a')$ come from the current Q-matrix, which is a mapping from a discrete state-action space to Q-values. α is the learning rate taking values between 0 and 1. It can be shown that Q-learning converges to the optimal Q-values with probability 1 as long as all actions are repeatedly sampled in all states and state-action pairs are discrete (Watkins and Dayan, 1992).

The Q-learning algorithm works well to find the optimal policy when the state-action space is small. However, it would become computationally inefficient and even infeasible to compute Q-values for every state-action pair when the state-action space is large (just imagine Eq. (5) needs to be repeatedly computed for a large number of state-action combinations, with constant updates of the Q-matrix). This is where deep learning can help reduce the computational burden. Specifically, a parameterized DNN can be integrated with an RL algorithm like Q-learning, to efficiently approximate the optimal Q-values instead of maintaining and updating a Q-matrix while applying Eq. (5).

More specifically, we adapt the DQN algorithm, proposed by Minh et al. (2015), to the problem considered in this paper. DQN is a relatively new DRL algorithm that uses a convolutional neural network as a function approximator of the Q-function. DQN has excelled in video game environments where the state-action space is very large. A prominent advantage of DQN is that it overcomes instability and divergence that occur when a nonlinear function approximator such as a neural network is used to represent the Q-function, by embedding two salient feature: experience replay and target network, whose use will be discussed in subsection 3.3. Before getting into the details of DQN, below we first describe our specifications of states, actions, and rewards in the context of crowdshipping.

3.2 DRL formulation for crowdshipping

3.2.1 Information array

In this section, we propose a novel state and action space design as well as reward function specification for crowdshipping. A key in this proposal is the creation of an information array that contains the routing sequence of each crowdsourcee. Let J and K denote respectively the sets of shipping requests and crowdsourcees. The information array is a $|K| \times (2|J| + 1)$ matrix where $|K|$ and $|J|$ denote respectively the numbers of crowdsourcees (which is equivalent to the number of routes) and shipping requests. Each row indicates the routing sequence of one crowdsourcee. The matrix has $2|J| + 1$ columns to accommodate the extreme possibility that all $|J|$ requests ($2|J|$ nodes) are assigned to a single crowdsourcee plus the origin node of the crowdsourcee (thus one more node needs to be added). For example, if the kth row of the information array contains the following tuple: $(u_k, p_1, p_2, d_1, d_2)$, it means that crowdsourcee k will leave his/her origin node u_k, go to the pickup node of the first request p_1, pick up the second request p_2, then drop off the first request d_1, and finally drop off the second request d_2. In this case, the cells of the first five columns of the kth row are occupied, whereas the remaining cells in the row are empty (Fig. 2).

	1	2	3	4	5	⋯	$2\|J\|+1$
1	u_1	⋯	⋯	⋯	⋯	⋯	⋯
⋮	⋯	⋯	⋯	⋯	⋯	⋯	⋯
k	u_k	p_1	p_2	d_1	d_2	0	0
⋮	⋯	⋯	⋯	⋯	⋯	⋯	⋯
$\|K\|$	$u_{\|K\|}$	⋯	⋯	⋯	⋯	⋯	⋯

Fig. 2. Illustration of the information array

The information array is constantly updated after every time step. Given that we consider a static problem for the purpose of operation planning rather than real-time decision support, we assume that all requests are unassigned (i.e., assigned to backup vehicles) at the beginning. Thus, initially each row in the information array contains only the origin node of a crowdsourcee.

3.2.2 State representation using a three-tuple

The information array provides a foundation for specifying the state space. At each time step t, the state of the crowdshipping environment is described by a three-tuple $s_t = \{S^l, S^r, S^c\}$ which provides respectively: 1) location information of pickup and delivery nodes of requests and crowdsourcee routing sequences; 2) request-specific time information; and 3) crowdsourcee-specific

time and capacity information. With $\{S^l, S^r, S^c\}$, the agent not only has a complete picture of the crowdsourcee routing sequences, but can leverage the time-related information to perform heuristics-guided actions, as described in subsection 3.2.3.

The first component in the three-tuple, S^l, is specified as follows:

$$S^l = \left\{n_i, n_j^p, n_j^d, n_k^c; \forall i \in J \cup K, j \in J, k \in K\right\}$$

where

- n_i is the coordinate of node i;
- n_j^p is the coordinate of the successor node of the pickup node of request j if j is assigned;
- n_j^d is the coordinate of the predecessor node of the delivery node of request j if j is assigned;
- n_k^c is the coordinate of the first node visited by crowdsourcee k if the crowdsourcee is assigned (i.e., first node other than the crowdsourcee origin).

The second component in the three-tuple, S^r, contains three pieces of request-specific time information:

$$S^r = \left\{\mathcal{s}_j, b_j, o_j; \forall j \in J\right\}$$

where

- $\mathcal{s}_j$ is the slack time of request j;
- b_j is the unused service time of request j;
- o_j is the occupation time of request j.

For a request j, slack time $\mathcal{s}_j$ measures how urgent it needs to be assigned:

$$\mathcal{s}_j = \begin{cases} \left(t_{d_j}^l - t_{p_j}^e\right) - T_{p_j,d_j}^c & f_j = 0 \\ \mathcal{M} & f_j = 1 \end{cases} \tag{6}$$

where

- $t_{d_j}^l$ is the latest delivery time for request j;
- $t_{p_j}^e$ is the earliest pickup time for request j;
- T_{p_j,d_j}^c is the direct travel time by crowdsourcee from pickup node p_j to delivery node d_j;
- $\mathcal{M}$ is a very large number;
- f_j equals 1 if request j is assigned to a crowdsourcee, and 0 otherwise.

For an unassigned request j, its urgency is the difference between the largest amount of time allowed for pickup and delivery ($t_{d_j}^l - t_{p_j}^e$), and the minimum amount of time needed to do so by crowdsourcee (T_{p_j,d_j}^c). The larger the difference, the lower the urgency with which the request needs

to be assigned. For an assigned request, a very large number $\mathcal{M}$ is given, which means that its urgency is effectively zero (as it is already assigned). Using this urgency measure, the agent solving for the assignments can prioritize assigning requests that have not been assigned to crowdsourcees.

The unused service time of a request j (b_j) quantifies the gap between the latest delivery time $t^l_{d_j}$ and the actual delivery time t_{d_j} (Eq. (7)). Conceptually, a larger b_j means greater flexibility in altering the way the request is picked up and delivered (e.g., by moving the request to a different position in the assigned crowdsourcee route or to a different route).

$$b_j = t^l_{d_j} - t_{d_j} \tag{7}$$

Note that in the case of an unassigned request, the request will be delivered by a backup vehicle which departs from a pre-specified depot D. Assuming that the backup vehicle will leave the depot at the earliest pickup time $t^e_{p_j}$, the actual delivery time will be $t_{d_j} = t^e_{p_j} + T^b_{D,p_j} + T^b_{p_j,d_j}$ where T^b_{D,p_j} and $T^b_{p_j,d_j}$ denote respectively the travel time of the backup vehicle from the depot to the pickup node, and from the pickup node directly to the delivery node.

The occupation time of a request (o_j) quantifies the duration between pickup and delivery of a request j.

$$o_j = t_{d_j} - t_{p_j} \tag{8}$$

where

t_{p_j} is the pickup time of request j by the assigned crowdsourcee.

For an unassigned request, t_{p_j} is equal to $t^e_{p_j} + T^b_{D,p_j}$. Thus, $o_j = T^b_{p_j,d_j}$.

The third component in the three-tuple, namely S^c, contains four pieces of crowdsourcee-specific time and capacity information:

$$S^c = \{\mathfrak{h}_k, v_k, \tau_k, \eta_k; \forall k \in K\}$$

where

$\mathfrak{h}_k$ is the routing duration for crowdsourcee k;
v_k is the total delivery time violation of requests assigned to crowdsourcee route k;
τ_k is the remaining available time for crowdsourcee k;
η_k is the total capacity violation along the route of crowdsourcee k.

The calculation of $\mathfrak{h}_k$ is intuitive. v_k is calculated using Eq. (9), where J^k denote the set of requests assigned to crowdsourcee k. The max operator is used when delivery is earlier than the latest delivery time (i.e., $t_{d_j} - t^l_{d_j} \leq 0$) such that it does not contribute to the violation:

$$v_k = \sum_{j \in J^k} \max(t_{d_j} - t^l_{d_j}, 0) \tag{9}$$

The remaining available time of crowdsourcee k, τ_k, is the difference between the crowdsourcee's total available time ($t^k_{\text{end}} - t^k_{\text{start}}$) and the route duration ($\mathfrak{h}_k$), as shown in Eq. (10), where t^k_{end} and t^k_{start} are the end and start of crowdsource k's available time window. An underlying assumption is that an assigned crowdsourcee will start routing at t^k_{start}. If the total available time of a crowdsourcee is less than the route duration, $\tau_k < 0$ means that crowdsourcee k's time availability is violated when finishing the last delivery on the route.

$$\tau_k = \left(t^k_{\text{end}} - t^k_{\text{start}}\right) - \mathfrak{h}_k. \tag{10}$$

Given that a crowdsourcee has limited carrying capacity (measured in weight), the total capacity violation along a crowdsourcee route η_k is the total number of capacity violation occurrences at each pickup node:

$$\eta_k = \sum_{j \in J^k} \delta_{p_j}, \tag{11}$$

where $\delta_{p_j} = 1$ if the total weight carried right after picking up at node p_j exceeds the carrying capacity, and zero otherwise.

With the full specification of $s_t = \{S^l, S^r, S^c\}$, the dimension of the state space can be explicitly expressed as a function of the dimensions of J and K. We show this in Remark 1 below.

Remark 1. The dimension of the state space is $11|J| + 8|K|$.

Proof. See Appendix A.

3.2.3 Action space design

As mentioned in Section 1, the combinatorial nature of the crowdshipping problem means that a large number of different actions can be taken to construct and improve crowdsourcee routing.

However, enumerating all possible actions would be neither efficient nor practical in DRL training. To preserve training tractability, we abstract the action space into five types of actions. At each time step, the agent may perform one action from the five types to alter an existing crowdsourcee route(s) or create a new crowdsourcee route. The choice of an action type is informed by the DQN algorithm. Once the action type is identified, the specific action to take is directed by the heuristics that leverage time-related state information about crowdsourcees and requests (e.g., slack time of a request and remaining available time of a crowdsourcee) so that assignment urgency, flexibility for routing improvement, and shipping cost reduction potential are taken into account toward more efficient crowdsourcee routing construction/improvement.

Among the five types of actions, the first type pertains to inserting an unassigned request to an existing/new route. The other three types of actions: intra-route move, inter-route move, and 1-exchange, are neighborhood moves of requests that have been previously placed in some existing crowdsourcee routes. Here the term "neighborhood" means that a move makes only one change to the solution, such that the solutions before and after the move remain quite similar to each other. Details of performing the neighborhood moves are described in subsections 3.2.3.2-3.2.3.4 below. The last action type is do-nothing, i.e., no action is taken. We consider do-nothing as an action for preserving good solutions. Specifically, if a very good solution has been achieved, having the option of do-nothing prevents taking another action that would move away from the solution to an inferior solution. It is worth mentioning that other more complex actions can be realized using the proposed five action types, in multiple time steps. In other words, our proposed action types are building blocks for other more complex actions. For instance, a 3-way exchange of requests among three crowdsourcee routes in a cyclic manner could be decomposed into and realized through two inter-route moves. Because of the decomposition, it is possible that the inter-route moves are taken consecutively or with other actions in between, therefore permitting more flexibility. Fig. 3 provides an illustration of the first four action types.

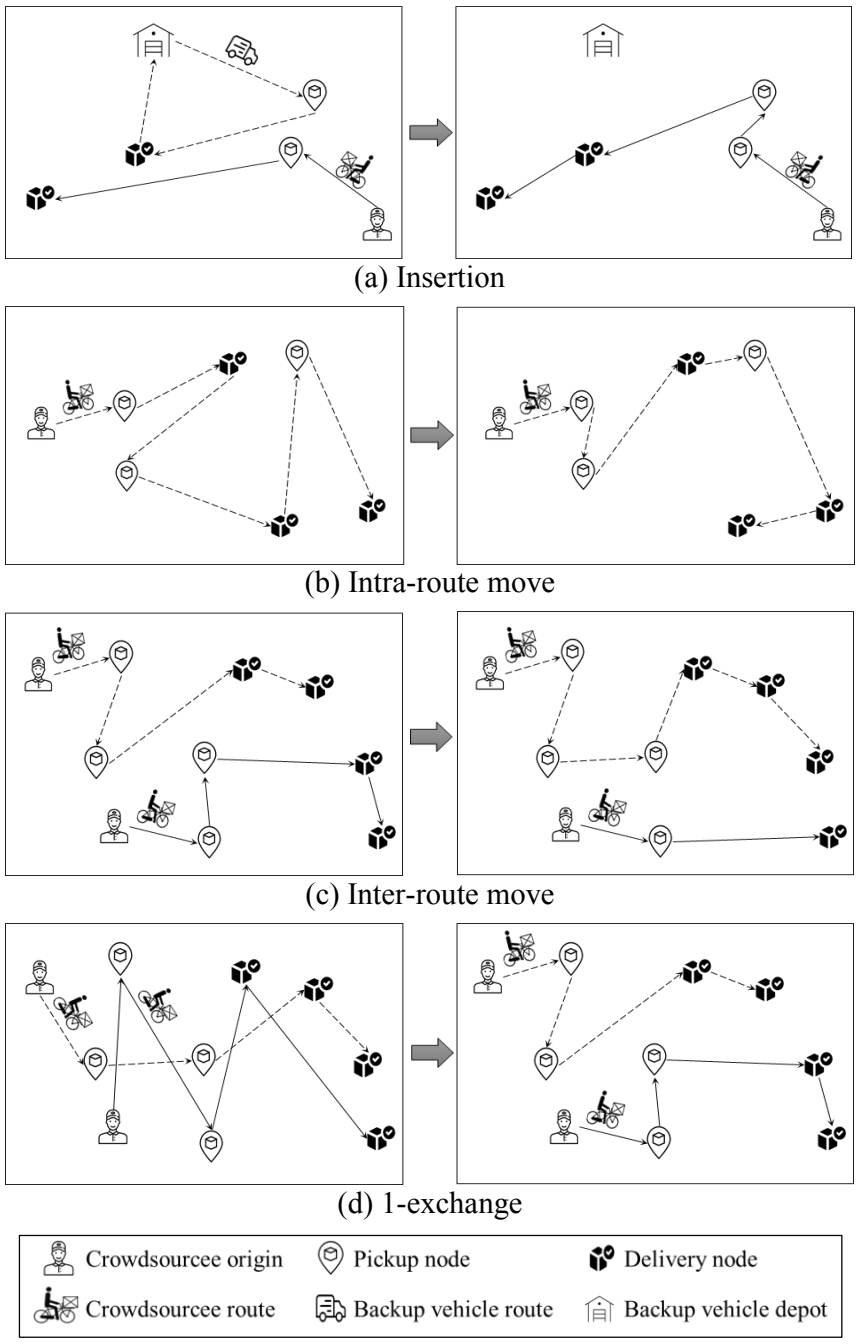


Fig. 3. Illustration of the four types of actions considered

For insertion, intra-route move, and inter-route move, routing feasibility after taking an action needs to be checked by following Definition 1 below.

Definition 1. Feasibility of a crowdsourcee route. A crowdsourcee route k is feasible if the following four conditions are met:

(1) request pickup is no earlier than the earliest pickup, for all requests on the route: $t_{p_j} \geq t_{p_j}^e, \forall j \in J^k$;

(2) request delivery is no later than the latest delivery, for all requests on the route: $t_{d_j} \leq t^l_{d_j}, \forall j \in J^k$;

(3) remaining available time of the crowdsourcee after completing the route is non-negative: $\tau_k \geq 0$;

(4) no violation of crowdsourcee capacity on the route: $\eta_k = 0$.

Conceptually, the decision on what action to take at a time step proceeds in two stages. First, the DQN algorithm identifies one of the five action types (insertion, intra-route move, inter-route move, 1-exchange, and do-nothing) as specified in the action space. Once the action type is identified, in the second stage the specific action is executed using the corresponding heuristic, based on system state information and intuitive reasonings. The remainder of this subsection describes in detail the heuristics that guide the specific action to take under each action type (except for do-nothing). We also present computational complexity of each heuristic as Remarks 1-4, with proofs provided in Appendix A.

3.2.3.1 Inserting an unassigned request in an existing/new route

For insertion, we need to determine which request to choose for insertion, and where to insert the request. The action consists of three steps.

Step 1: *Select a request.*

Among the unassigned requests, select one with the smallest slack time.

Step 2: *Insert the request to a route.*

For the selected request, calculate the distances between the pickup node of the request and each crowdsourcee. For an assigned crowdsourcee, the distance is to the end of the crowdsourcee route. For an unassigned crowdsourcee, the distance is to the crowdsourcee origin. Identify the smallest distance.

If the smallest distance occurs to an assigned crowdsourcee, insert the node to the end of the crowdsourcee route. If the smallest distance occurs to an idle crowdsourcee, create a new route: crowdsourcee origin → request pickup node → request delivery node.

Step 3: *Perform intra-route move.*

If the request is inserted to an existing crowdsourcee route, explore moving the request to earlier positions in the route. The move follows Step 2 of intra-route move below, but for the inserted request only. Place the request at the position that is feasible and leads to the smallest routing cost.

If it is not feasible to place the request anywhere in the inserted route, move to the route with the second smallest distance, insert the request to the end of the route, and perform intra-route move. If all routes are checked and a feasible placement cannot be found, move to the unassigned request with the second smallest slack time. Repeat Step 2 and Step 3 described above. If it is not possible to feasibly insert any unassigned request, stop and nothing is changed.

In Step 1, the rationale for considering the unassigned request with the smallest slack time, the information of which comes from s_j in S^r (second component in the three-tuple state representation), is that we want to get the most urgent unassigned request assigned first. In Step 2, we perform insertion to the nearest crowdsourcee as this incurs the smallest time loss between the crowdsourcee finishing the currently assigned requests and picking up the request under study.

Remark 2. The computational complexity of insertion is $O(|J|^2 \log|J|)$.

Proof. See Appendix A.

3.2.3.2 Intra-route move

Intra-route move involves moving a later request to an earlier position in a route to reduce routing cost. The action also consists of three steps.

Step 1: *Select a route.*

Select the crowdsourcee route with the largest remaining available time.

Step 2: *Move examination.*

Enumerate all feasible moves of a request to a different place. For a request, first move it to the end of the route, i.e., having the last two nodes in a route as the pickup and delivery nodes of the request. Then, examine all feasible moves of the request to an earlier place in the route.

To illustrate, consider routing sequence $(u_k, p_1, d_1, \dots, p_{n-1}, d_{n-1}, p_n, d_n)$ and moving request n (whose pickup and delivery nodes are already at the end of the route). Move p_n to an earlier position, one place at a time, i.e., to the places right before d_{n-1}, right before p_{n-1},…, until right after u_k. For each new position of p_n, examine feasibility of holding d_n at its initial place, moving it one place at a time to an earlier position, as long as d_n is not before p_n. For each feasible (p_n, d_n) move, calculate the routing cost.

Repeat the above for every request in the route.

Step 3: *Identify the best move.*

Among all the feasible moves in Step 2, pick the one with the smallest routing cost. If the routing cost is smaller than the original routing cost, perform the move. If no move yields a smaller routing cost or there is no feasible move, stop and nothing is changed.

In Step 1, the rationale for considering the route with the largest remaining available time, for which the information comes from τ_k in S^c, is that such a route has the greatest flexibility for moving requests around.

Remark 3. The computational complexity of intra-route move is $O(|J| \log|J|)$.

Proof. See Appendix A.

3.2.3.3 Inter-route move

Inter-route move picks a request from a route and moves it to another route by performing the following three steps.

Step 1: *Select a request.*

Among all assigned requests, select one with the largest occupation time.

Step 2: *Move the request to the end of a different route.*

Insert request to another existing route or create a new route, following Step 2 of insertion. Calculate the combined routing cost for the two routes involved in the inter-route move.

Step 3: *Perform intra-route move of the request.*

If the request is inserted to an existing crowdsourcee route, explore moving the pickup and delivery nodes of the request to earlier positions in the route to reduce routing cost. The move follows Step 2 of intra-route move below, but for the inserted request only.

If there exist feasible intra-route moves that lead to lower routing cost than the cost after Step 2, perform the intra-route move that leads to the lowest routing cost. If not and the solution after Step 2 is feasible, perform only Steps 1-2. Otherwise, nothing is changed.

In Step 1, the rationale for considering the request with the largest occupation time, for which the information comes from o_j in S^r, is that larger occupation time may suggest greater time (and thus cost) reduction potential by moving the request to a different route.

Remark 4. The computational complexity of inter-route move is $O(|J|\log|J|)$.

Proof. See Appendix A.

3.2.3.4 1-exchange move

1-exchange move pertains to exchanging two requests which are on two crowdsourcee routes. Performing the move has four steps.

Step 1: *Select the first request.*

Among all assigned requests, select the first request that has the largest unused service time.

Step 2: *Select the second request.*

Excluding the route associated with the first selected request, select the second request that has the largest unused service time among the remaining assigned requests.

Step 3: *Exchange the selected requests.*

Remove the two requests from their routes. Add each request to the end of the other route. Calculate the combined routing cost for the two routes involved in the 1-exchange move.

Step 4: *Perform intra-route move of the two requests.*

For each of the two requests, this follows Step 2 of intra-route move. Place the request at the position that is feasible and leads to the smallest routing cost. The associated routing cost should be lower than the routing cost from Step 3. If this is not possible, leave the request at the end of the route, i.e., do not perform Step 4.

In Steps 1 and 2, the rationale for choosing requests with the largest unused service time, for which the information comes from b_j in S^r, is that such requests have the greatest flexibility to be moved around. It should be noted that unlike the other three actions, we do not consider feasibility while performing Step 3 in 1-exchange. This is intentional to help the search escape local optima (Nanry and Barnes, 2000).

Remark 5. The computational complexity of 1-exchange move is $O(|J|\log|J|)$.

Proof. See Appendix A.

3.2.4 Reward specification

Given the state and the action at a time step, we specify the reward as the change in TSC as a result of the action taken. If the action taken at time step t is inserting request j in crowdsourcee route k, the reward is computed as:

$$r_t = \beta^c \mathfrak{h}_{k,t-1} + \beta^b \left(T^b_{D,p_j} + T^b_{p_j,d_j} + s + T^b_{d_j,D}\right) - \beta^c \mathfrak{h}_{k,t} \tag{12}$$

where

β^b is the unit cost of using a backup vehicle (in \$/minute);
β^c is the unit cost of using a crowdsourcee (in \$/minute);
$\mathfrak{h}_{k,t-1}$ is the route duration (in minutes) of crowdsourcee route k at time step $t-1$;
$\mathfrak{h}_{k,t}$ is the route duration (in minutes) of crowdsourcee route k at time step t;
s is the stopping time of a node (assumed one minutes);
$T^b_{d_j,D}$ is the backup vehicle travel time from the delivery node of request j back to depot D.

In Eq. (12), $\beta^c \mathfrak{h}_{k,t-1}$ is the cost of crowdsourcee route k at time step $t-1$, which is before request j is inserted. If the route does not exist before inserting j, this term will be zero. $\beta^b(T^b_{D,p_j} + T^b_{p_j,d_j} +$

$T^b_{d_j,D}$) is the cost of picking up and delivering the request by a backup vehicle. $\beta^c \mathfrak{h}_{k,t}$ is the cost of the crowdsourcee route k at time step t, after request j is inserted. The calculation result using Eq. (12) is measured in dollars.

If the action taken at time step t is a neighborhood move, let us use Ψ_t to denote the set of crowdsourcee route(s) that are involved in the move. For intra-route move, Ψ_t will have just one route. For inter-route move and 1-exchange, Ψ_t will have two routes. The reward is calculated as the difference of the routing costs before and after the move:

$$r_t = c_t^1 - c_t^2 \tag{13}$$

where c_t^1 and c_t^2 are routing costs for the route(s) in Ψ_t before and after the neighborhood move:

$$c_t^1 = \beta^c \left(\sum_{k \in \Psi_t} \mathfrak{h}_{k,t-1} + \vartheta \sum_{k \in \Psi_t} v_{k,t-1} + \tau \sum_{k \in \Psi_t} \chi_{k,t-1} + \rho\phi \sum_{k \in \Psi_t} \eta_{k,t-1} \right) \tag{14}$$

$$c_t^2 = \beta^c \left(\sum_{k \in \Psi_t} \mathfrak{h}_{k,t} + \vartheta \sum_{k \in \Psi_t} v_{k,t} + \tau \sum_{k \in \Psi_t} \chi_{k,t} + \rho\phi \sum_{k \in \Psi_t} \eta_{k,t} \right) \tag{15}$$

where

$\mathfrak{h}_{k,t}$	is the route duration for crowdsourcee k at time step t;
$v_{k,t}$	is the delivery time violation of requests assigned to crowdsourcee route k at time step t;
$\chi_{k,t}$	is the available time violation for crowdsourcee k at time step t;
$\eta_{k,t}$	is the carrying capacity violation for crowdsourcee k at time step t;
ϑ	is the penalty multiplier for delivery time violation;
τ	is the penalty multiplier for crowdsourcee overworking;
ρ	is the penalty multiplier for crowdsourcee carrying capacity violation;
ϕ	is the capacity violation-to-time conversion factor.

Here,ϑ, τ, and ρ are unitless penalty parameters. ϕ has unit of minutes per capacity violation. The calculation result of Eq. (13) is also in dollars.

3.3 DRL algorithm for crowdshipping

This subsection describes how DQN, which is our training algorithm, is adapted to the context of crowdshipping. DQN is an off-policy RL approach, as it is based on Q-learning (Sutton and Barto, 2018). The training is offline with a simulator developed by ourselves. In DQN, the training of the agent is through multiple episodes, each. Each episode is associated with a crowdshipping problem

instance of a certain size, which is randomly generated and starts with an initial state that all crowdsourcees are idle (unassigned). Training in an episode involves improving the solution by taking actions described in subsection 3.2.3, one at a time in a number of time steps.

At each time step, an ε-greedy strategy is employed to consider both exploration and exploitation as the agent decides what type of action to take among insertion, intra-route move, inter-route move, 1-exchange move, and do-nothing. By exploration, it means that the agent takes a random action type, with probability ε. By exploitation, the agent takes one of the five action types above that is the best—based on the experiences that the agent has learned so far (reflected in the current Q-values, as shown in line 7 in Algorithm 1 at the end of this subsection), with probability $1-\varepsilon$. Once the best action type is chosen, the specific action follows the heuristics described in subsection 3.2.3 (line 8 in Algorithm 1). Consequently, a reward and a new state are observed.

While exploitation takes advantage of what have been learned in terms of the best action to take, exploration is necessary to try to get the agent out of local optima toward even better action sequences, to further reduce total shipping cost. At the beginning of an episode, ε takes value 1, i.e., the focus is purely on exploration, which is intuitive as the agent has zero learned experience (thus nothing to exploit) at this point. Then as time goes by, the agent gradually increases the probability of exploiting learned actions. A decay rate of ξ is used which describes the change in probabilities between two time steps (Eq. (16)). ξ is a hyper parameter.

$$\varepsilon_{t+1} = \varepsilon_t(1-\xi) \tag{16}$$

One salient feature of DQN is experience replay, for which a replay memory M is used to store the agent's experiences during training. Up to $|M|$ experiences can be stored in the replay memory. An experience is associated with taking an action at a given state and time step, observing a state transition, and getting a reward. For example, at time step t, the agent performs an action a_t which transforms the state from s_t to s_{t+1} and yields a reward r_t. The experience is denoted as $e_t = (s_t, a_{t,} r_t, s_{t+1})$. At the beginning of the training, M is empty. As the training continues, experiences are accumulated and added to replay memory M. Once $|M|$ experiences are stored in M, adding a new experience requires simultaneous removal of the oldest experience stored in M.

At each time step, a DNN is trained using a minibatch M_{sub} of samples that are randomly selected from M. Note that in the beginning of the training, the number of accumulated experiences in M will be fewer than $|M_{\text{sub}}|$. In this case, experiences will continuously be accumulated in M but DNN will not be trained, until the replay memory has $|M_{\text{sub}}|$ experiences. The employment of experience replay

using randomly selected minibatch samples has multiple advantages. First, because the samples are randomly selected, correlation between samples will be less than learning directly from consecutive samples, thereby enhancing the efficiency of learning. Second, each experience can potentially be used in many weight updates, thus allowing for greater data efficiency. Third, by experience replay the behavior distribution is averaged over many previous states, which contributes to smoothing out learning and avoiding oscillation or divergence in the parameters (Mnih et al., 2015).

For each selected experience (s, a, r, s'), state s is used as the input for the DNN (with weight parameters θ) to generate state-action value $Q(s, a: \theta)$, or Q-value, which is the output of the DNN.[2] Collectively for all the selected experiences, the prediction of $Q(s, a: \theta)$'s comprises the first *forward pass*. $Q(s, a: \theta)$ is then compared with the target optimal Q-value $Q_*(s, a)$ which gives the maximum expected return achievable by following any DQN policy. Ideally, the target optimal Q-value should satisfy the Bellman optimality equation:

$$Q_*(s, a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a' \in A} Q_*(s', a') \middle| s, a\right] \tag{17}$$

where r is the immediate reward by taking action a at station s.

The comparison of $Q(s, a: \theta)$ with $Q_*(s, a)$ is performed using a loss function. Assuming a square form for the loss function and replacing $Q_*(s, a)$ by the RHS of Eq. (17), the loss function $\mathcal{L}$, which depends on DNN weight parameters θ, can be expressed as:

$$\mathcal{L}(\theta) = \frac{1}{|M_{\text{sub}}|} \sum_{(s,a,r,s') \in M_{\text{sub}}} \left[r + \gamma \max_{a' \in A} Q_*(s', a') - Q(s, a: \theta)\right]^2 \tag{18}$$

Obviously, to calculate $\mathcal{L}(\theta)$, $Q_*(s', a')$ is needed. However, $Q_*(s', a')$ is unknown (if $Q_*(s', a')$ was known, then the training would be done). One way to get an approximation of $Q_*(s', a')$ is to perform another *forward pass* with the DNN, i.e., for state s' in each experience (s, a, r, s') along with the same weight parameters θ of the DNN, predict state-action values $Q(s', a': \theta), \forall a' \in A$ using the DNN. By approximating $Q_*(s', a')$ with $Q(s', a': \theta)$, Eq. (17) can be re-expressed as:

$$\mathcal{L}(\theta) = \frac{1}{|M_{\text{sub}}|} \sum_{(s,a,r,s') \in M_{\text{sub}}} \left[r + \gamma \max_{a' \in A} Q(s', a': \theta) - Q(s, a: \theta)\right]^2 \tag{19}$$

[2] In this paper, we use an architecture in which there is a separate output for each possible action. Only the state is the input to the DNN. Thus, among the outputs for different actions, we choose the one corresponding to action a as $Q(s, a: \theta)$.

After performing two forward passes as described above, the gradient of the loss in Eq. (19) is used to update θ by the Adam optimizer, a widely-used gradient descent-based algorithm for minimizing the loss (Kingma and Ba, 2015). However, a main drawback exists in this two-forward pass procedure. When θ gets updated, the Q-values obtained from this network will also get updated (in the next time step). So will the target Q-values as they are calculated using the same network parameter. In other words, the direction of updates for the Q-values and the target Q-values will be same. As a consequence, the correlation between the Q-values and the target Q-values can be high, possibly leading to oscillation or divergence of the policy during training.

To tackle this issue, a second salient feature of DQN is that a parallel network, called the target network, of the original DNN is created to preserve DNN parameter values for a period of time, so that target Q-values do not get updated with the same frequency as the Q-values. The target network, which is a clone of the original network, initializes its parameters θ' using the original DNN: $\theta' = \theta$ at the beginning of the training. Then, instead of updating θ' by θ of the original DNN at every time step, θ' is frozen for δ time steps. Only after every δ time steps, θ' gets updated to whatever is the present value of the original network parameters θ. In this procedure, δ is a hyper parameter.

The Q-value obtained from the target network $Q(s', a': \theta')$ is used to calculate the approximate target Q-value $r + \gamma \max_{a' \in A} Q(s', a': \theta')$. The loss function shown in Eq. (17) becomes:

$$\mathcal{L}(\theta) = \frac{1}{|M_{\text{sub}}|} \sum_{(s,a,r,s') \in M_{\text{sub}}} \left[r + \gamma \max_{a' \in A} Q(s', a': \theta') - Q(s, a: \theta) \right]^2 \tag{20}$$

In implementing DQN, we use a slightly modified version of the squared loss function called Huber loss function. For each sample, the squared term is used only if the absolute error falls below a threshold (here we choose the value 1). Otherwise, we use an absolute term as shown in Eq. (21). An advantage of the Huber function form is that the loss is less sensitive to outliers than the square loss for large errors, which prevents exploding gradients.

$$\mathcal{L}(\theta) = \frac{1}{|M_{\text{sub}}|} \sum_{(s,a,r,s') \in M_{\text{sub}}} L_H \left(r + \gamma \max_{a' \in A} Q(s', a': \theta') - Q(s, a: \theta) \right) \tag{21}$$

where

$$L_H\left(r+\gamma\max_{a'\in A}Q(s',a':\theta')-Q(s,a:\theta)\right)=\begin{cases}0.5\left[r+\gamma\max_{a'\in A}Q(s',a':\theta')-Q(s,a:\theta)\right]^2 \\ \qquad\qquad \text{if } \left|r+\gamma\max_{a'\in A}Q(s',a':\theta')-Q(s,a:\theta)\right|<1 \\ \left|r+\gamma\max_{a'\in A}Q(s',a':\theta')-Q(s,a:\theta)\right|-0.5 \text{ otherwise}\end{cases}$$

Finally, it should be mentioned that during an episode, we also accumulate the rewards that are negative. If the accumulated negative reward in an episode falls below a threshold, then the training of the episode is perceived as not promising and consequently terminates.

Summarizing, the overall learning algorithm is presented in Algorithm 1 and illustrated in Fig. 4 below.

Algorithm 1: Overall learning algorithm for the crowdshipping problem

Initialize replay memory $M = \emptyset$
Initialize the original DNN with random weight parameters θ
Initialize the target DNN with same structure as the original DNN and weight parameters $\theta' = \theta$
for episode $i = 1$ to I, **do** ▷ I is the number of episodes
    Initialize state $s_0 \in S$ ▷ in the initial state s_0, all crowdsourcees are unassigned
    for time step $t = 1$ to T, **do** ▷ T is the number of time steps in an episode
        Select a random action type a_t with probability ε; otherwise, set action type $a_t = \operatorname{argmax}_{a\in A} Q(s_t, a; \theta)$
        Execute a specific action under action type a_t, as guided by the corresponding heuristic in subsection 3.2.3. This results in r_t and s_{t+1}
        Store experience $e_t = (s_t, a_t, r_t, s_{t+1})$ in M
        if $|M| > |M_{\text{sub}}|$, **do**
            if $\mathbb{R}_t > \mathcal{K}$, **do** ▷ $\mathbb{R}_t$ is accumulated negative reward in the episode; $\mathcal{K}$ is a threshold
                Randomly sample a minibatch M_{sub} of experiences from M
                for each experience $e_j = (s_j, a_j, r_j, s_{j+1})$ in M_{sub}, **do**
                    Compute $r_j + \gamma \max_{a'\in A} Q(s_{j+1}, a':\theta')$
                end for
                Calculate loss by Eq. (21)
                Update weight parameters θ by the Adam optimizer
                Update $\theta' = \theta$ every δ time steps
            else
                break ▷ if $\mathbb{R}_t < \mathcal{K}$, the training is perceived as not promising and stop
            end if
        end if
    end for
end for

Based on Algorithm 1 and the state and action space characterization, the complexity of DQN training is examined, with results formalized as Remark 6 below.

Remark 6. The computational complexity of DQN training in the context of crowdshipping is $O(\{([(|J|+\ell)e+\ell e^2]|M_{\text{sub}}|+|J|^2\log|J|)\}IT)$, where ℓ is the number of hidden layers in the DNN, e is the upper bound on the number of neurons in a hidden layer, $|M_{\text{sub}}|$ is minibatch size, I is the number of episodes, and T is the upper bound on the number of time steps in an episode (note in line 20 of Algorithm 1 that training in an episode can stop earlier).

Proof. See Appendix A.

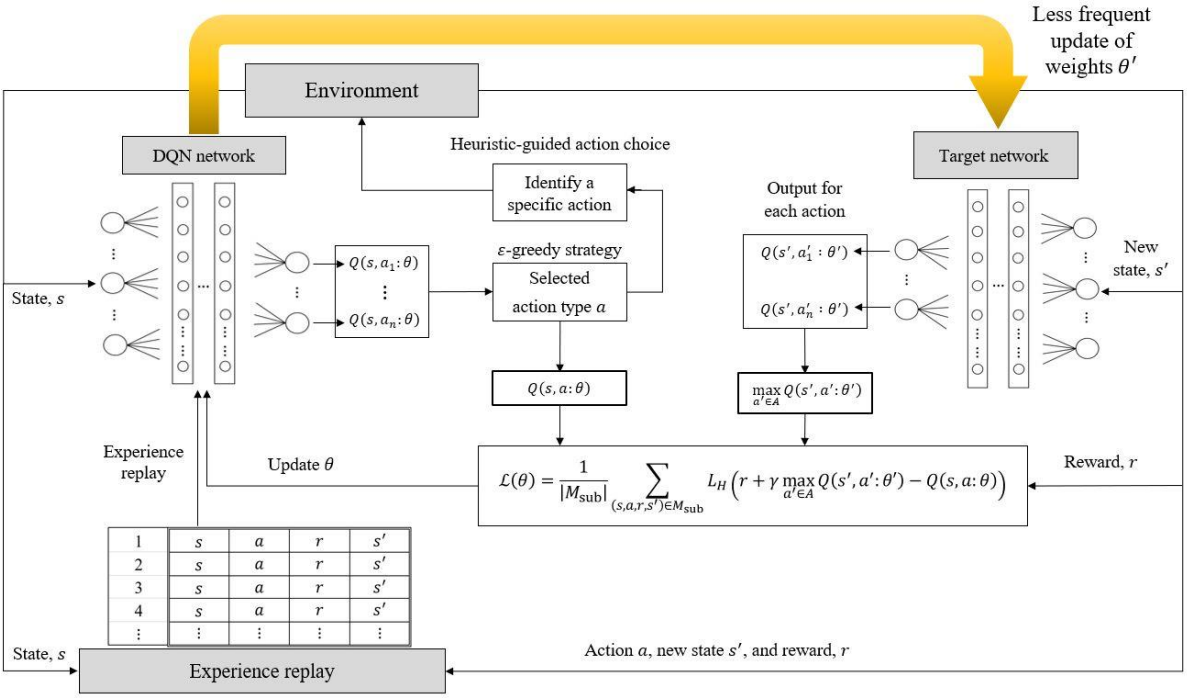


Fig. 4. The architecture of the overall learning algorithm

3.4 Rule-interposing in DRL training and implementation

Whether in DRL training or in implementation of the trained policy, it is possible that some routes or node sequences are repeatedly visited during neighborhood moves. This reduces the efficiency of DRL training, as well as the efficiency in search for the best crowdsourcee-request assignment outcome when a trained policy is applied to solve a problem instance (note that after DRL training is done, at a given state s the optimal Q-value only provides what type of action to take, i.e., $a^*=\underset{a\in A}{\text{argmax}}\,Q^*(s,a)$ where Q^* denotes the optimal Q-value). In this subsection, we propose two rules that aim to prevent such repeated visiting of routes and node sequences, by excluding a previously visited route or node

sequence from being considered again in a number of subsequent actions. In what follows, the first rule focuses on routes. The second rule focuses on node sequences.

3.4.1 Rule 1: Introducing priority lists for route selection

To avoid that actions are repeatedly exerted on one or a subset of crowdsourcee routes, the first rule proposed relies on construction and use of three priority lists of crowdsourcee routes, with each list corresponding to one of the three neighborhood move action types (intra-route move, inter-route move, and 1-exchange) described in subsection 3.2.3. Specifically, when a neighborhood move action type is chosen, we pick the crowdsourcee route(s) from the top of the corresponding priority list to apply the action type. After the specific action is taken on the route(s), the route(s) are removed from the list. Thus over time, routes will be continuously picked and removed from the priority list. The priority list will be shortened, and eventually become empty. Then, we construct a new priority list of all the crowdsourcee routes for the same action type. By doing so, during the life cycle of a priority list, a route is considered only once for the associated action type. This allows more exploration of the same action type on other routes. This construction-destruction of priority lists repeats throughout the training and implementation of a trained DRL to solve a problem instance.

Each of the three priority lists is constructed based on some criterion. For intra-route move, the priority list is constructed by sorting crowdsourcee routes in descending order based on the crowdsourcee's remaining available time, which is consistent with the rationale of Step 1 in subsection 3.2.3.2. For inter-route move, the priority list is constructed by sorting crowdsourcee routes in descending order based on the occupation time of each crowdsourcee, measured as the duration between the time of the last delivery and the time of the first pickup. This is in line with Step 1 in subsection 3.2.3.3 (there we also consider the largest occupation time, though for requests). Thus, request selection of Step 1 in subsection 3.2.3.3 will be only from the route with the highest priority. After an inter-route move action is taken, that route is removed from the priority list. The occupation time of the route to which the request is moved will be updated. The position of that route in the priority list will also be updated, for which the computational complexity is $O(\log(N))$ based on binary search, with N being the number of crowdsourcee routes in the priority list. For 1-exchange move, the priority list is constructed by sorting crowdsourcee routes in descending order based on unused service time, which is consistent with Steps 1 and 2 in subsection 3.2.3.4. Thus, the selection of the first and the second requests will be from the two routes with the highest and second highest priority respectively. After a 1-exchange move action is taken, the two routes will be removed from the priority list.

3.4.2 Rule 2: Imposing Tabu tenure for neighborhood moves

The second interposing rule is that after a request node (either pickup or delivery) is moved away from an adjacent node (either right before or right after in the routing sequence) on a crowdsourcee route, the former node cannot be moved back to the same location relative to the latter node over a certain number of subsequent actions. This latter node can be of a different request, or of the same request (i.e., the former node is the pickup node of a request, and the latter node is the delivery node of the same request). Similar to Rule 1, this rule applies to the three types of neighborhood moves (intra-route move, inter-route move, and 1-exchange). For each type of neighborhood move, a Tabu tenure will be created to record for how many subsequent actions a request node cannot be neighbored with another node. Similar to Rule 1, Tabu tenure allows neighborhood moves to explore more routing sequences, rather than getting trapped in routing sequences that have been explored and only locally optimal.

To operationalize Tabu tenure, two matrices are created and maintained. The first matrix, of dimension $2|J| \times 2|J|$, indicates whether a node (indexed by the column. There are in total $|J|$ requests thus $2|J|$ pickup and delivery nodes) preceding another node (indexed by the row) is Tabu-ed and for how long. The second matrix, of dimension $(|K| + 2|J|) \times 2|J|$, indicates whether a node (indexed by the column) following another node (indexed by the row) is Tabu-ed and for how long. The second matrix has $|K|$ more rows which correspond to the origins of the $|K|$ crowdsourcees, as a pickup node can be placed right after the origin of a crowdsourcee. Given that the two matrices are relatively sparse, we adopt a three-coordinate representation that records only the row number, column number, and value of the non-zero elements (indicating for how many subsequent actions a position is Tabu-ed) of a sparse matrix, rather than storing the entire matrix. For example, suppose we deal with the preceding relationships of two requests ($|J| = 2$), which leads to a matrix of 4×4. If node 1 (pickup node of request 1) preceding node 3 (pickup node of request 2) is Tabu-ed for the next two subsequent actions, and node 2 (delivery node of request 1) preceding node 4 (delivery node of request 2) is Tabu-ed for the next three subsequent actions, we record only two elements in each of three lists: Row list: [3, 4]; Column list: [1, 2]; and Data list: [2, 3]. In our numerical experiments in Section 4, this dealing with sparse matrices is shown to significantly reduce DRL training time (by 14%). The precedence/ succession relationships are updated whenever an action is taken. Note that if a Tabu-ed position yields a solution that is better than the best solution obtained so far for a problem, then the Tabu tenure will be overridden.

4 Numerical experiments

This section illustrates numerical implementation of the proposed methodology described in Section 3. We primarily investigate two problem sizes: a medium size with 50 requests and 22 crowdsourcees, and a larger size with 200 requests and 70 crowdsourcees. In subsection 4.1, we first present and discuss the results for the medium-size problems in detail, including problem setup, training results, results comparison with full, partial, and no time-related information in the state space, benefits of heuristics-guided action choice and rule-interposing, and results sensitivity to key hyperparameters. To further gauge the performance of the DRL-based approach, benchmarking is performed in subsection 4.2. This includes comparison with three popular heuristic methods as well as with optimal solutions that can be obtained for a series of small-size problem instances. In subsection 4.3, to keep the paper length we briefly report implementation results for the larger-size problem instances in terms of total shipping cost and computation time, in comparison with the three heuristics. The DQN algorithm is coded and trained in the PyTorch environment. All numerical investigations are conducted on a PC with Intel Core i9-10920X CPUs at 3.50GHz and 128GB RAM and NVIDIA Titan RTX GPUs.

4.1 Medium-size problems: DRL training and application

4.1.1 Setup

As mentioned above, we consider a static problem of assigning 50 requests to 22 crowdsourcees. Following Remark 1, the dimension of the state space is $11 \times 50 + 8 \times 22 = 726$. The service area has a square shape of 6 miles $\times$ 6 miles. For a problem instance in both training and testing, the pickup and delivery locations of each request are randomly generated in the service area. So are the origins of the crowdsourcees. The available time of a crowdsourcee is randomly drawn from a uniform distribution of 1-2 hours. The weight of a shipping request is also randomly drawn from a uniform distribution of 2-7 lbs. The carrying capacity of a crowdsourcee is 10 lbs. The earliest pickup time of all requests is the present time. The latest delivery time of a request is randomly drawn from a uniform distribution of 100-120 minutes. Crowdsourcees are assumed to bike to perform pickup and delivery at a speed of 10 mph. Given that all problem instances are randomly generated, in the statistical sense there should be no difference between instances for training and testing. On the other hand, as a very large number of instances are used (e.g., 808 instances used in training), for illustration we only present two randomly picked instances, one from training and one from testing, as shown in Appendix B. No significant differences (apart from the effect due to randomness) among the instances can be discerned.

If a request is not assigned to any crowdsourcees, it will be picked up and delivered by a backup vehicle which leaves a depot located at the center of the service area and returns to the depot after finishing the delivery. Given the small weight of a request relative to the typical carrying capacity of a backup vehicle, capacity constraints are not considered for backup vehicles. We assume backup vehicles travel at a speed of 20 mph. We follow Kafle et al. (2017) by setting the operating cost of a backup vehicle to be \$68/hour (\$1.13/minute) and the pay rate for crowdsourcees to be \$10/hour (\$0.17/minute), which is considerably cheaper. Crowdsourcees get paid whenever carrying requests.

Following Mnih et al. (2015), values of hyperparameters, shown in Table 2, are selected by performing an informal search. It should be noted that since these hyperparameter values are chosen for our specific crowdshipping problems, they may not be the best hyperparameter values for other problem settings. Nonetheless, if a new problem setting bears similarities with our crowdshipping problems (e.g., a pickup and delivery problem with capacity constraints only or without constraints), the hyperparameter values identified here could be a good start point for hyperparameter fine tuning. We set $T = 85$ time steps as the upper bound on the length of an episode, the penalty parameters in the reward specification to be $\vartheta = 0.1$, $\tau = 0.2$, and $\rho\phi = 0.15$ minutes/capacity violation, and the length of Tabu tenure to be three consecutive actions. We choose a 6-layer (i.e., $\ell = 6$) fully connected feed-forward neural network as our DNN construction, where each hidden layer has 512 neurons (i.e., $e = 512$).

Table 2: Hyperparameter values

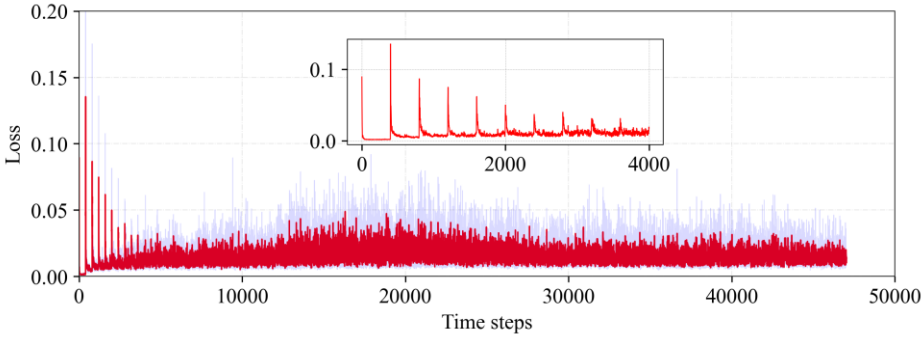
Hyperparameter	**Value**
Replay memory size ($\|M\|$)	15,000
Minibatch size ($\|M_{\text{sub}}\|$)	256
Target network update frequency (δ)	400
Discount factor (γ)	0.88
Learning rate (α)	0.0001
Decay rate (ξ)	0.001
Episode termination threshold ($\mathcal{K}$)	-55

4.1.2 Training results

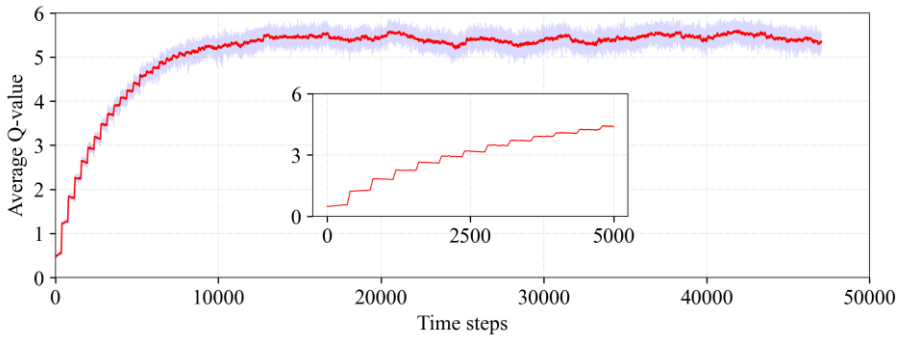
Fig. 5 plots the evolution of training over time steps, using four measures: (a) loss per time step. The light purple curve reflects the actual values, while the red curve is the running average over three time steps; (b) average Q-value, averaged over all state-action pairs in a minibatch. Similar to (a), the light purple curve reflects the actual values, while the red curve is the running average over 65 time steps; (c) accumulated reward between two terminations. As described in subsection 3.3, a termination

occurs when the accumulated negative reward falls below threshold $\mathcal{K}$ (line 20 of Algorithm 1); (d) cumulative penalty, which is the sum of the last three terms in the parentheses (multiplied by β^c) in Eq. (14)-(15) over all time steps from the start of the training. In (a) and (b), running averages are taken to better illustrate the changes and trends. The training stops when the relative change in cumulative penalty in the most recent 3,000 time steps is less than 5%. In total, 46,071 time steps (778 episodes) are used in the DQN training. The training takes 49.3 minutes.

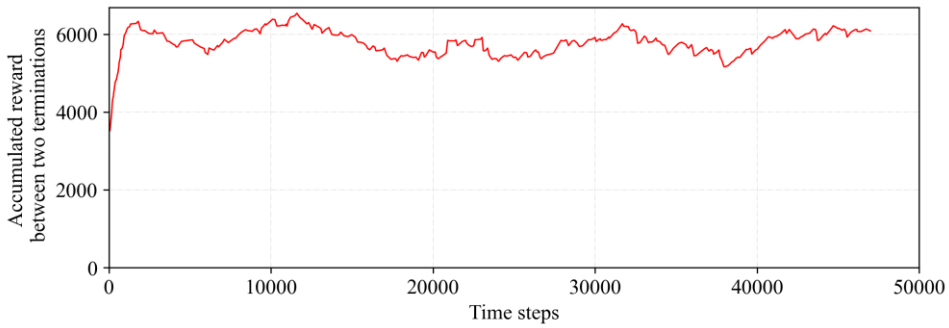
Fig. 5(a) illustrates that in the early stage of training, the loss value per time step experiences a jump every 400 time steps (as made clearer in the zoom-in view), which corresponds to an update of the target network. The jumps are particularly acute in the beginning since the agent has little learned experience then. The magnitude of jumps diminishes as learning continues. In Fig. 5(b), the average Q-value keeps improving till after 13,500 time steps. Before that, updates in the DNN considerably improve the DQN algorithm which yields better solutions. Fig. 5(b) also shows a magnifier of the first 5,000 time steps. It is interesting to observe step-wise jumps every 400 time steps, which is again the target network update frequency. In other words, whenever updating the target network, it leads to a significant improvement in average Q-value. The magnitude of the jumps decreases over time steps, suggesting that the marginal improvement of the DQN algorithm is diminishing as training continues. In Fig. 5(c), the accumulated reward between two terminations tends to stabilize after around 42,000 time steps. Fig. 5(d) shows that the accumulative penalty over all time steps becomes stable a bit later: after around 38,000 time steps, the DQN algorithm becomes well trained that taking actions suggested by the DQN algorithm will cause little violation of time and capacity constraints (which incurs penalty) during neighborhood moves.



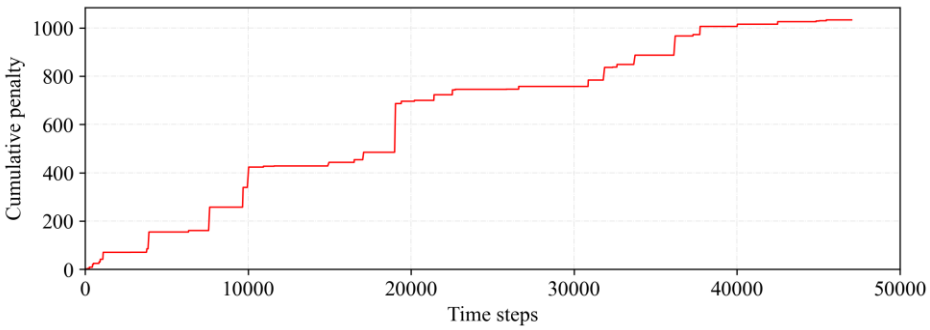
(a)



(b)



(c)



(d)

Fig. 5. Evolution of (a) loss; (b) average Q-value; (c) accumulated reward between two terminations; and (d) cumulative penalty in the course of DQN training

To further show the effectiveness of the DQN algorithm training, we apply the DQN algorithm throughout its training to three randomly generated problem instances of the same size (50 requests and 22 crowdsourcees). Fig. 6 shows the TSC results when applying the DQN algorithm with the most up-to-date DNN weight parameters every 40 episodes. It can be seen that TSC will be drastically reduced after the first 40 episodes. For example, for problem instance 1 TSC reduces by more than three-quarters from 1,475 to less than 250. Afterwards, the improvement in TSC is more incremental with some rebounds. After 640 episodes, TSC becomes very stable for all three problem instances (as shown further in the zoom-in view).

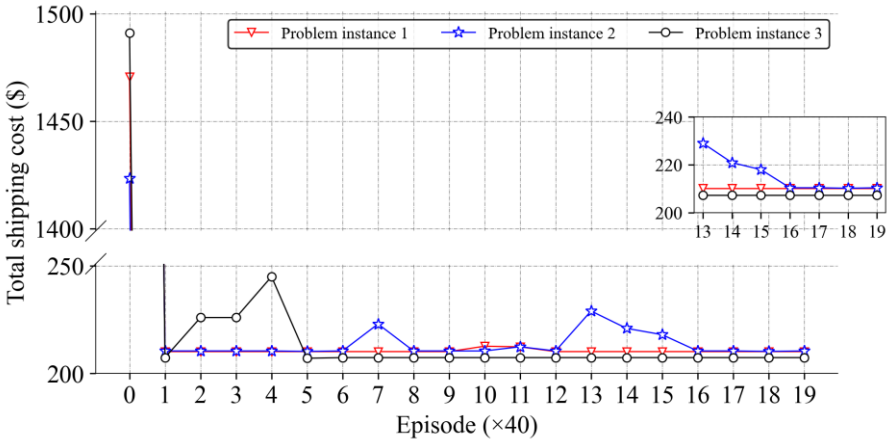


Fig. 6. Evolution of total shipping cost during training

4.1.3 Assessing the benefits of heuristics-guided action choice

Recall that one novelty of our proposed DRL algorithm lies in the embedment of heuristics-guided action choice in DRL. At each time step, the DRL agent performs one of the five types of actions to create new or change existing crowdsourcee routes. To compare the proposed DRL algorithm with a DRL algorithm without heuristics-guided action choice, a neighborhood move will be randomly chosen given any of the first four action types, as described in Appendix C. Similar to what we do in Fig. 6, we apply the DQN algorithm throughout its training to two randomly generated problem instances and present the TSC results using the most up-to-date DNN weight parameters every 40 episodes. For each problem instance, we train the DQN algorithm twice, one with heuristics-guided action choice and the other without. The results are shown in Fig. 7.

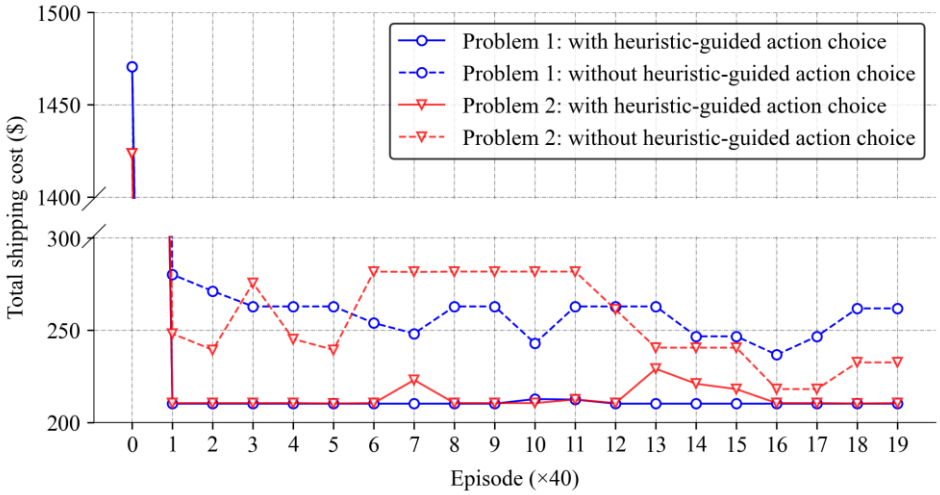


Fig. 7. Comparison of total shipping cost with and without heuristics-guided action choice during training

We observe that at the beginning of the training, the TSC curve with heuristics-guided action choice (solid line) is higher than without heuristics-guided action choice (dashed line) for both problem instances. However, for the rest of training, the TSC curves with heuristics-guided action choice are well below the TSC curves without heuristics-guided action choice. At the end of training, a substantial TSC gap remains. The final TSC without heuristics-guided action choice is 24.5 and 10.5% higher than with heuristics-guided action choice, for the two problem instances respectively. The results clearly show the advantage of heuristics-guided action choice in DQN training.

Fig. 8 presents comparisons of applying the trained DRL models to 20 randomly generated problem instances. The reduction of TSC with heuristics-guided action choice is clearly observed. Across the 20 problem instances, the average TSC reduction is 11.4% with a standard deviation of 9.6%. The largest reduction, which occurs to problem instance 8, is 35.5%.

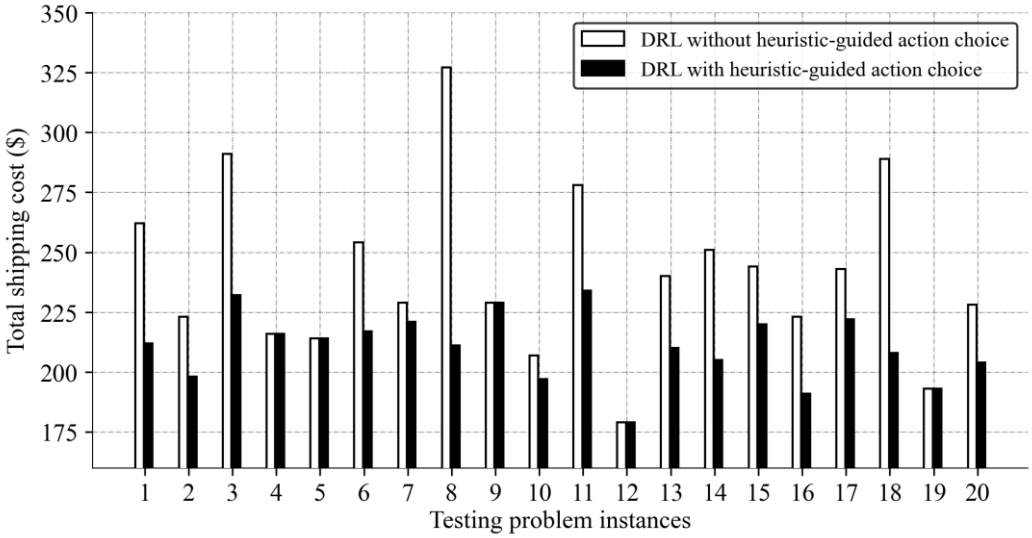


Fig. 8. Comparison of total shipping cost with and without heuristics-guided action choice during testing

Fig. 9 reports further the DQN training time without and with heuristics-guided action choice. To make sensible comparisons, we let training without heuristics-guided action choice run same number of time steps. The results show that the training time with heuristics-guided action choice (49.3 minutes) is much larger than without heuristics-guided action choice (29.6 minutes). This suggests that a non-trivial amount of added computation is needed during training for heuristics-guided action choice to achieve lower TSC.

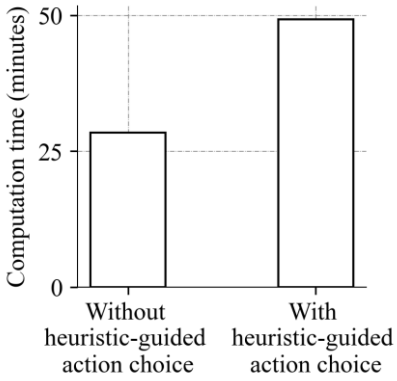


Fig. 9. Comparison of training time with and without heuristics-guided action choice

4.1.4 Assessing the benefits of rule-interposing

In this subsection we evaluate the benefits of another novelty of the proposed DRL algorithm: the integration of rule-interposing into DRL training and implementation. As in Fig. 7, we apply the DQN algorithm throughout its training to the same two randomly generated problem instances, and present the TSC results using the most up-to-date DNN weight parameters every 40 episodes. For each problem instance, we train the DQN algorithm twice, one with rule-interposing and the other without. The results are shown in Fig. 10.

For the first problem instance (in blue), although the TSC without the two rules appears to be diminishing at the beginning of the training, the TSC value rebounds after 80 episodes, then declines and meets the TSC curve when the two rules are used at around 240 episodes. Afterwards, the TSC curve without the two rules experiences some fluctuations, surges after around 560 episodes, and remains well above the TSC curve with the two rules. At the end of the training, the TSC without the two rules is 24.3% higher than with the rules. For the second problem instance (in red), the TSC curve without the two rules experiences greater fluctuations throughout the episodes. Overall, the results also demonstrate the advantage of rule-interposing in DQN training.

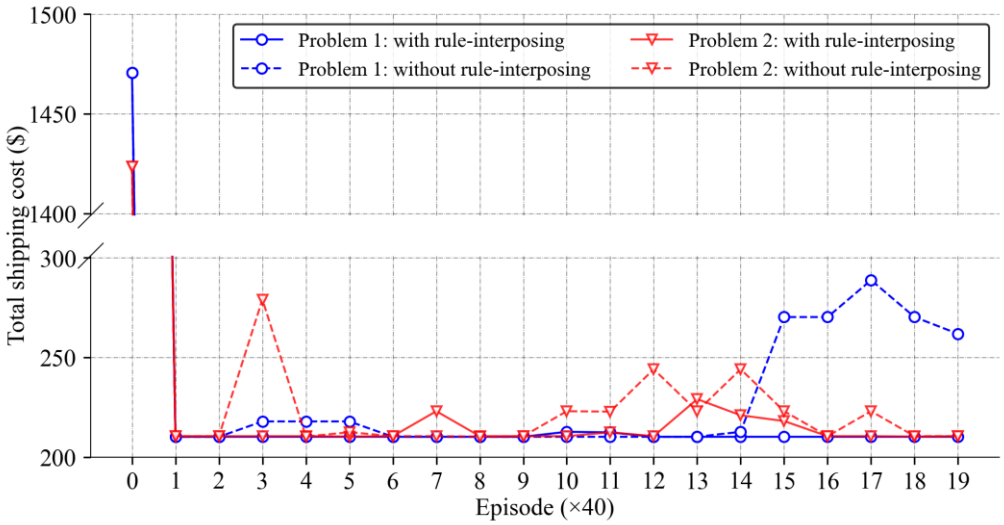


Fig. 10. Comparison of total shipping cost with and without rule-interposing

Fig. 11 presents comparisons of applying the trained DRL models to the same 20 randomly generated problem instances as in subsection 4.1.3. We observe an overall TSC reduction with rule-interposing. The average TSC reduction across all 20 instances is 9.2% with a standard deviation of 12.1%. The largest reduction, which occurs to problem instance 17, is 34.5%.

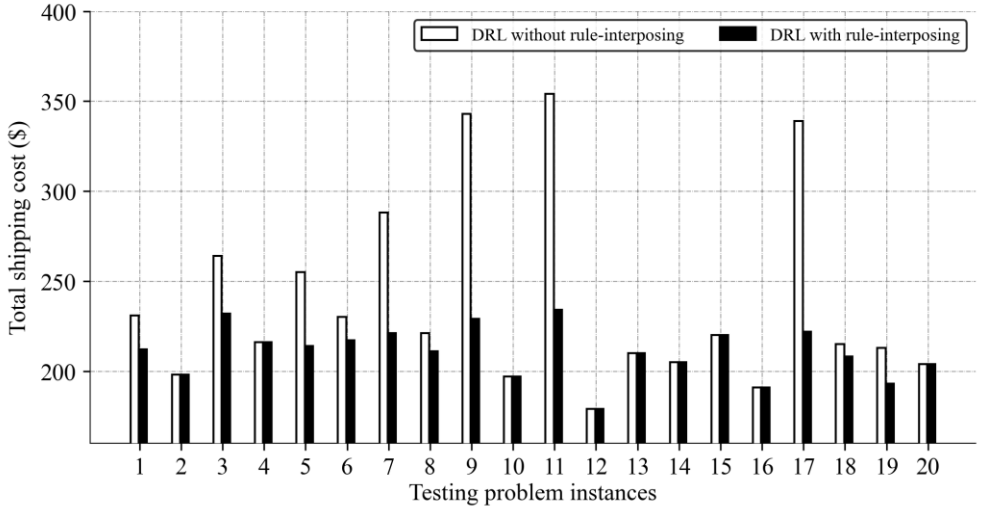


Fig. 11. Comparison of total shipping cost with and without rule-interposing during testing

Fig. 12 reports further the DQN training time with and without the two rules embedded. Again, to make sensible comparisons, we allow the training without the two rules to run the same number of time steps. The results show that the training time with rule-interposing (49.3 minutes) is only slightly higher than without rule-interposing (47.1 minutes).

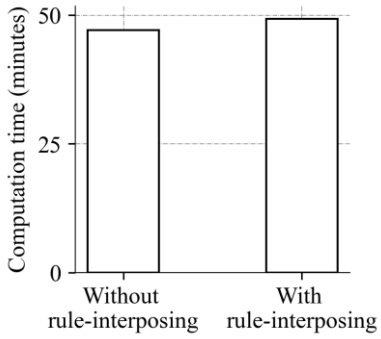


Fig. 12. Comparison of training time with and without rule-interposing

4.1.5 Comparison with full, partial, and no time-related information in state space

A uniqueness of the state representation is the specification and inclusion of a variety of time information that relate to both requests and crowdsourcees. A question arises as to how important such information is in training the DRL agent. To this end, this subsection investigates the possibility of having lower-dimension state space representation without part or all of the time-related information. Specifically, three alternatives are investigated. The first alternative does not have request-related time information, that is, we remove S^r from the three-tuple state representation $s_t = \{S^l, S^r, S^c\}$. The second alternative does not have crowdsourcee-related time information, that is, we remove S^c from the three-tuple except for η_k which records violation of crowdsourcee carrying capacity. The third alternative is an additive of the first two alternatives, i.e., the state space does not include any time information related to requests and crowdsourcees. As a result of absent time information, a step in a heuristic that is directed by time information will be performed randomly. For example, if request-related information is removed, step 1 of insertion (subsection 3.2.3.1) would randomly select an unassigned request, rather than selecting the request with the smallest slack time.

A DRL model is trained under each of the three alternatives, and then applied to 20 randomly generated problem instances along with the DRL model trained with full time-related information as in subsection 3.2.2. Fig. 13 below reports the results. It can be seen that time information plays a crucial role in guiding crowdsourcee route construction and improvement to reduce total shipping cost. Without any or with only partial time information, the total shipping cost would be higher—in many problem instances significantly. The results clearly suggest the importance to have the full time-related information while characterizing the state space.

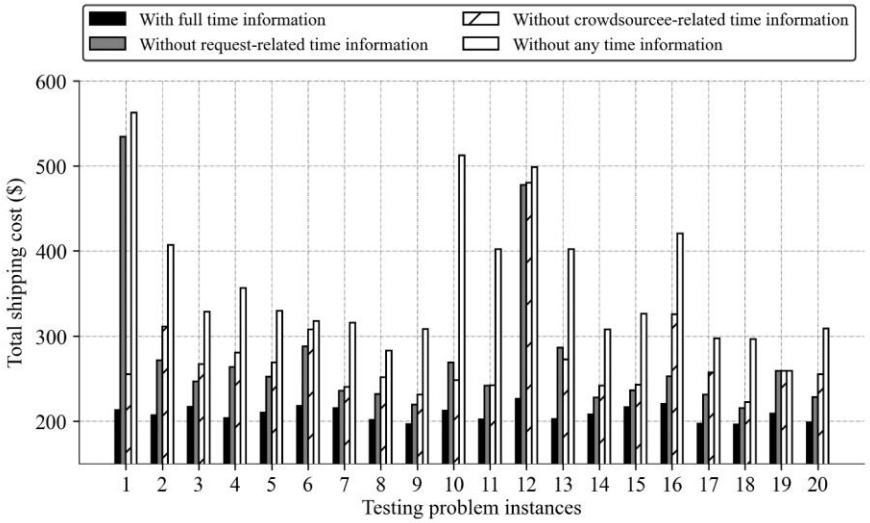
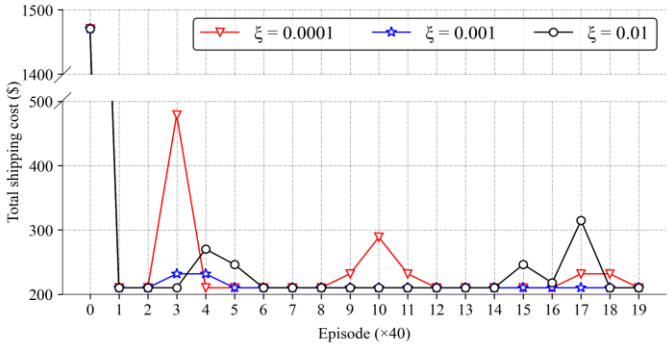


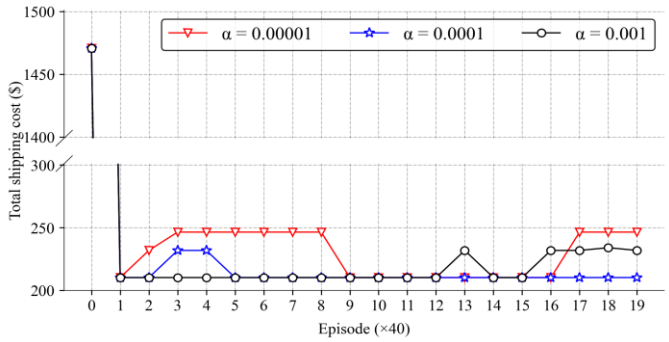
Fig. 13. Comparison of total shipping cost with full, partial, and no time-related information in the state space

4.1.6 Sensitivity of DQN training to hyperparameter values

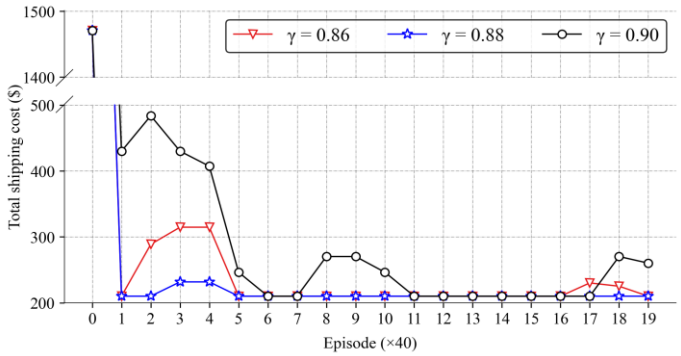
Finally, we investigate the sensitivity of DQN training to the values of four key hyperparameters: (a) decay rate ξ; (b) learning rate α; (c) discount factor γ; (d) target network update frequency δ. Fig. 14 presents the results. In each graph in Fig. 14, a curve corresponds to a specific value of the hyperparameter under investigation and is obtained in a similar fashion as the curves in Fig. 6, for a randomly generated problem instance. For a given graph, the three other hyperparameters not investigated in the graph take their values in Table 2. While Fig. 14 reports TSC values of one problem instance, we have also experimented with many other randomly generated problem instances and found consistent results. It can be seen that, for all graphs in Fig. 14, the chosen value for each hyperparameter produces more stable TSC curves than the alternative values. In addition, the final TSC using the chosen hyperparameter value is always no worse than using alternative values, which reaffirms our choice of the hyperparameter values.



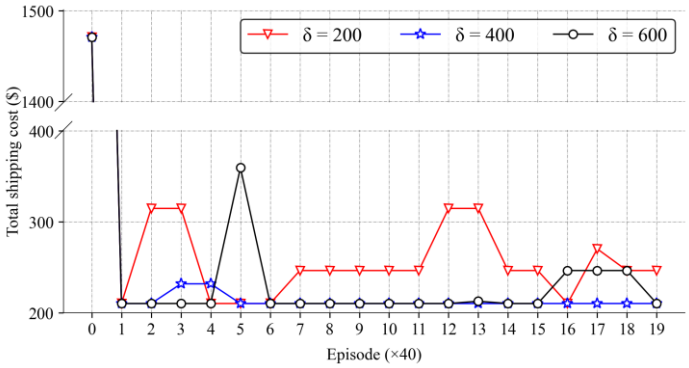
(a)



(b)



(c)



(d)

Fig. 14. Sensitivity of total shipping cost to different hyperparameter values: (a) ξ; (b) α; (c) γ; (d) δ

4.2 Benchmarking

To further gauge the performance of the DRL-based approach, benchmarking is performed in this subsection. We first compare the DRL-based approach with three popular heuristic methods: simple heuristic, reactive Tabu search (RTS), and simulated annealing (SA), for problem instances of the same size as in subsection 4.1. We also compare the DRL-based approach against optimal solutions, which come from formulating the problem as a mixed-integer linear program (MILP) and solving the MILP by CPLEX. The sizes of the problem instances are smaller so that optimal solutions can be obtained

884 within a reasonable amount of time (as is shown in subsection 4.2.2, even for such small sizes CPLEX
885 still cannot yield a solution for some instances).

886 ### 4.2.1 Comparison with heuristic methods

887 Among the three heuristic methods, the simple heuristic basically performs Steps 1-2 of the
888 insertion action described in subsection 3.2.3.1, and can generate solutions very fast. However, it does
889 not explore neighborhood moves. Therefore, the resulting solution can be far from optimum. RTS is a
890 hierarchical heuristic that dynamically adjusts search parameters and alternates between different
891 neighborhoods while seeking the optimal routing solution, based on the state and quality of the search.
892 Our implementation of RTS follows Nanry and Barnes (2000) with consideration of three types of
893 neighborhood moves (intra-route move, inter-route move, and 1-exchange). SA is based on the analogy
894 between the simulation of solids annealing and the problem of solving large combinatorial optimization
895 problems (Kirkpatrick et al., 1983; van Laarhoven and Aarts, 1987). Prior research shows that SA can
896 yield reasonably good solutions for large VRP instances and can be faster than other heuristics such as
897 Tabu search and genetic algorithm (Tan et al., 2001). At each temperature during cooling, an intra-
898 route move, an inter-route move, and an 1-exchange move as described in Ahamed and Zou (2020) are
899 performed in sequence, with each move followed by an evaluation that accepts not only an improved
900 solution, but also an inferior solution with certain probability. The parameter setting of SA follows
901 those in Kafle et al. (2017).

902 For the simple heuristic, it terminates when all feasible insertions of requests are performed. In
903 implementing RTS and SA, we allow for a sufficient number of iterations until the reduction in TSC
904 is not visible (TSC change is less than 2% in the last ten iterations). Fig. 15 presents the TSC results
905 using DRL and the three heuristics, for 20 randomly generated problem instances. DRL yields the best
906 solution in 18 out of the 20 problem instances. In contrast, the solutions using the simple heuristic are
907 the worst, despite small computation time as shown in Fig. 16. On the other hand, while the TSC results
908 from RTS and SA are closer to those using DRL, the computation time is much longer, by more than
909 an order of magnitude (20-40 minutes vs. mostly less than 1 second). Considering both solution quality
910 and time, the comparison clearly indicates the superiority of DRL.
911

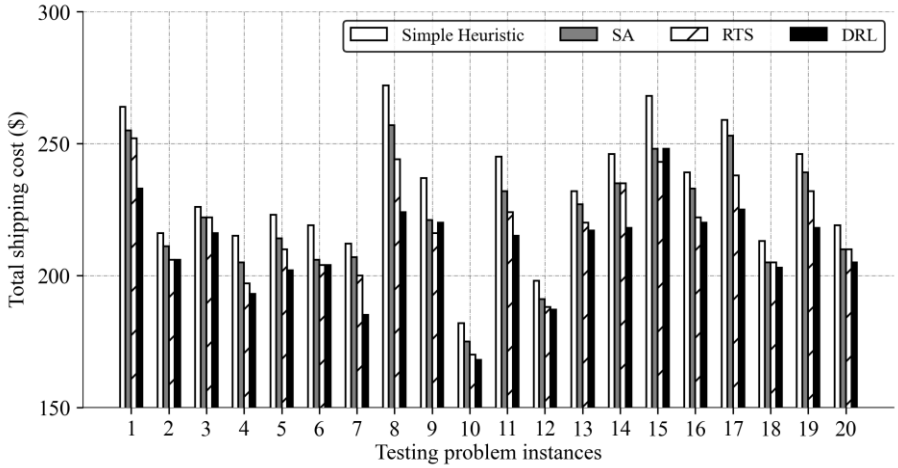


Fig. 15. Comparison of DRL with existing heuristics in terms of TSC (medium-size problems)

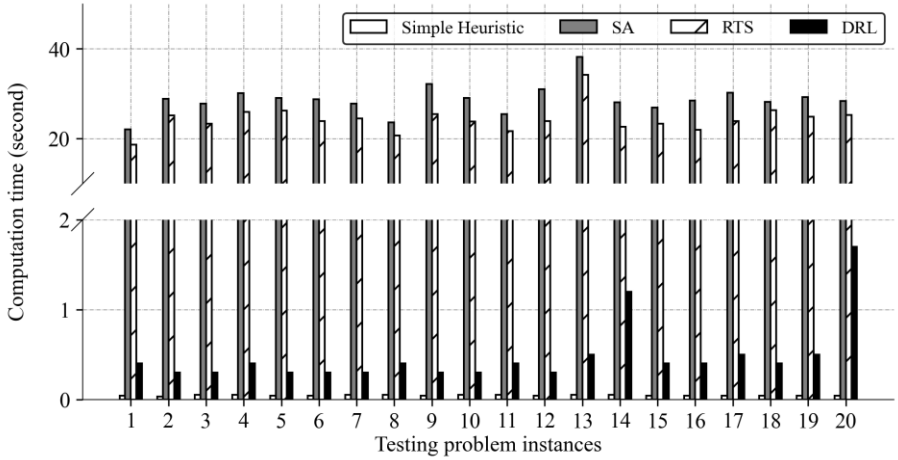


Fig. 16. Comparison of DRL with existing heuristics in computation time (medium-size problems)

4.2.2 Benchmarking with optimal solutions

To further investigate how close the solutions obtained from our approach are from the optimal solutions, we create two sets of benchmarking problem instances following the same procedure described in subsection 4.1.1. However, these instances have smaller sizes: problem instances in the first set each have six requests and three crowdsourcees. Problem instances in the second set each have eight requests and four crowdsourcees. We consider the smaller-size instances so that they can be solved to optimality using commercial solvers in a reasonable amount of time. These problem instances are deposited in GitHub for potentially further use by other researchers.[3]

The comparison results are shown in Tables 3-4. The 20 problem instances in Table 3 are those from the first set. The 15 problem instances in Table 4 correspond to the second set. Appendix D

[3] https://github.com/tahame2/DRL_benchmarking_2021.git

928 presents a Hamiltonian tour-based MILP formulation for the problems, which is solved by CPLEX
929 12.8 using the branch-and-bound method. Each problem instance is also solved by DRL. We report
930 both TSC and computation time using CPLEX and DRL. The last column in each table shows the
931 optimality gap (%), calculated as $\frac{\mathrm{TSC_{DRL}}-\mathrm{TSC_{CPLEX}}}{\mathrm{TSC_{CPLEX}}} \times 100\%$, where $\mathrm{TSC_{DRL}}$ is the TSC value from DRL
932 and $\mathrm{TSC_{CPLEX}}$ is the TSC value using CPLEX.
933 For the first set of problem instances, the optimality gap is between -3% and 16%, with an average
934 of 5.4%. We note that this average is comparable with some reported average gaps using DRL (e.g., in
935 Nazari et al., 2018), though their context is solving general VRP rather than crowdshipping problems.
936 The negative optimality gap is because the solution produced by CPLEX may not be exactly optimal
937 due to: 1) gap tolerance (the difference the best upper and lower bounds); and 2) integrality toleration
938 for integer variables. The occurrence of the negative gap and very small positive gap suggests that in
939 those instances DRL can yield solutions that are very close to the exact optimal solutions. On the other
940 hand, the computation time using CPLEX is much longer, with an average of 12.7 seconds, as
941 compared to 0.07 seconds by DRL. A similar conclusion can be made for the second set of problem
942 instances in Table 4, with an average optimality gap of 6.2%. It should be noted that with the slight
943 increase in problem size, the computation time by CPLEX has increased substantially, from an average
944 of 12.7 seconds to 16.6 minutes. In contrast, the average computation time by DRL remains at 0.07
945 seconds, suggesting strong scalability of the DRL approach.
946

Table 3: Comparison of solving 20 randomly generated problem instances each with 6 requests and 3 crowdsourcees using CPLEX and DRL

Problem instances	CPLEX		DRL		Optimality Gap (%)
	Total shipping cost ($)	Computation time (second)	Total shipping cost ($)	Computation time (second)	
P_6_3_1	29.6	27.0	31.5	0.04	6.4
P_6_3_2	24.6	3.4	24.3	0.10	-1.2
P_6_3_3	27.8	5.7	29.2	0.09	5.0
P_6_3_4	28.0	40.0	30.4	0.06	8.6
P_6_3_5	30.8	6.0	30.7	0.08	-0.3
P_6_3_6	25.8	4.0	25.8	0.07	0.0
P_6_3_7	26.8	3.0	29.6	0.07	10.4
P_6_3_8	27.8	9.5	27.2	0.07	-2.2
P_6_3_9	21.4	3.3	20.8	0.07	-2.8
P_6_3_10	25.5	30.2	28.2	0.07	10.6
P_6_3_11	29.6	3.7	30.9	0.06	4.4
P_6_3_12	24.5	7.0	27.9	0.05	13.9
P_6_3_13	15.4	2.5	17.2	0.05	11.7
P_6_3_14	25.7	2.9	28.0	0.07	8.9
P_6_3_15	19.5	3.0	20.9	0.07	7.2
P_6_3_16	37.4	54.5	36.6	0.06	-2.1
P_6_3_17	19.5	3.3	19.4	0.06	-0.5
P_6_3_18	21.2	2.5	22.2	0.06	4.7
P_6_3_19	25.9	39.6	30.1	0.06	16.2
P_6_3_20	21.1	3.0	22.9	0.06	8.5
Average		**12.7**		**0.07**	**5.4**

Table 4: Comparison of solving 15 randomly generated problem instances each with 8 requests and 4 crowdsourcees using CPLEX and DRL

Problem instances	CPLEX		DRL		Optimality Gap (%)
	Total shipping cost ($)	Computation time (min)	Total shipping cost ($)	Computation time (second)	
P_8_4_1	37.1	27.9	39.9	0.07	7.5
P_8_4_2	36.2	124.1	37.6	0.07	3.9
P_8_4_3	39.6	3.5	42.9	0.07	8.3
P_8_4_4	27.2	1.1	29.8	0.06	9.6
P_8_4_5	27.6	7.3	31.3	0.06	13.4
P_8_4_6	31.9	3.9	33.2	0.07	4.1
P_8_4_7	31.6	6.5	33.2	0.07	5.1
P_8_4_8	31.8	7.1	34.1	0.06	7.2
P_8_4_9	36.6	29.2	39.5	0.07	7.9
P_8_4_10	29.6	4.1	29.5	0.06	-0.3
P_8_4_11	26.0	1.4	25.6	0.09	-1.5
P_8_4_12	35.7	31.5	37.3	0.07	4.5
P_8_4_13	29.6	0.8	32.7	0.08	10.5
P_8_4_14	27.7	0.1	27.7	0.06	0.0
P_8_4_15	24.4	0.8	27.7	0.05	13.5
Average		**16.6**		**0.07**	**6.2**

4.3 Larger-size problems

4.3.1 Setup

The larger-size problem instance considers problems of assigning 200 requests to 70 crowdsourcees, which are of comparable size to many pickup-and-delivery operation planning problems investigated in the existing literature (Liu et al., 2015; Braekers and Kovacs, 2016; Ghilas et al., 2016). Apart from a larger number of requests and crowdsourcees, other setups and problem instance generation are the same as in the medium-size problems. With a larger problem size, it is natural to expect a higher number of time steps per episode to insert all requests and perform neighborhood moves of the requests. Therefore, we increase the length of an episode to 300 time steps. Following a similar informal search as in subsection 4.1.1, the penalty parameters in the reward specification are set to be $\vartheta = 0.25$, $\tau = 0.15$, and $\rho\phi = 0.2$, and the length of Tabu tenure to be 12 subsequent actions. The episode termination threshold $\mathcal{K}$ is decreased to -175. Decay rate ξ is set as 0.002. Other hyperparameter values remain the same. The training time takes 3 hours and 22 minutes.

4.3.2 Comparison of solutions using DRL and heuristics

We compare performance of the DRL-based approach with the three same heuristics as in subsection 4.2.1. 20 problem instances with 200 requests and 70 crowdsourcees are randomly generated. Fig. 17 shows that DRL yields the best solution in 18 out of the 20 instances. Again, the solutions from the simple heuristic are always the worst, despite small computation time (Fig. 18). While the resulting TSC values from RTS and SA are closer to those from DRL, the computation time is much longer (between 15-20 minutes vs. 2-3 seconds in most cases by DRL). By comparing the change in computation time from the medium-size problem (Fig. 14), it is clear that DRL is much more scalable than RTS or SA.

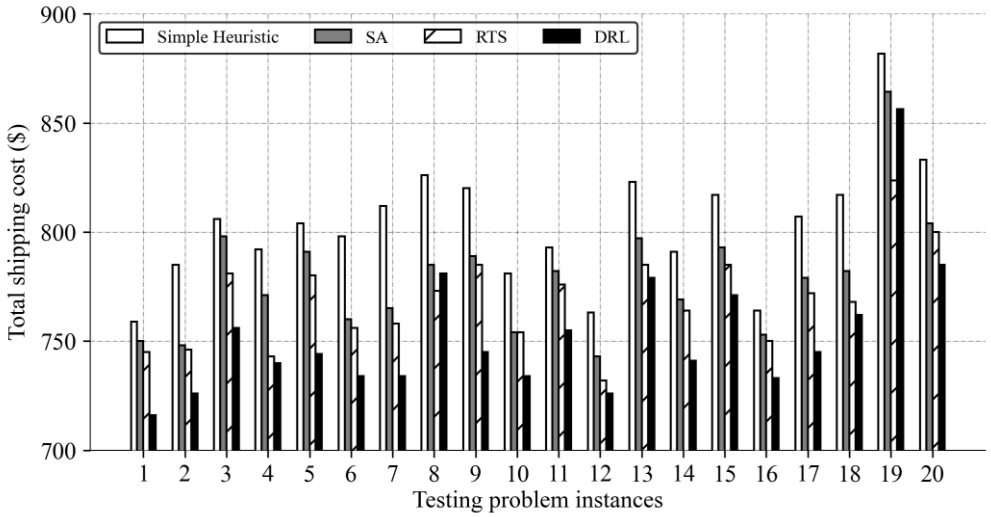


Fig. 17. Comparison of DRL with existing heuristics in terms of TSC (larger-size problems)

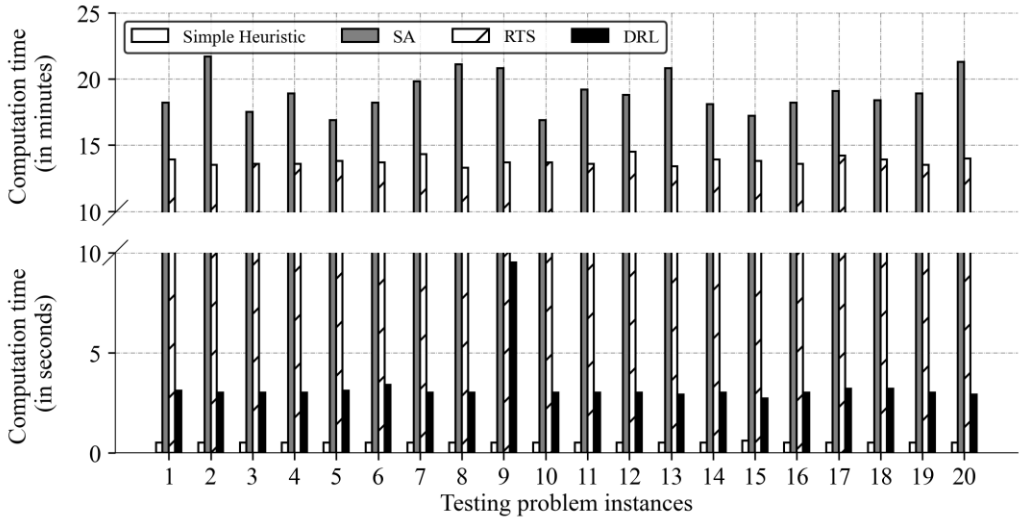


Fig. 18. Comparison of DRL with existing heuristics in computation time (larger-size problems)

5 Conclusion

Crowdshipping has gained increasing popularity for urban delivery given the low cost of hiring *ad hoc* couriers to perform pickups and deliveries. In this paper, we propose a novel, deep reinforcement learning-based approach to seek high-quality and computationally efficient assignment of requests to crowdsourcees. In performing the assignment, we consider that requests have time windows for pickup and delivery. In addition, crowdsourcees have limited time availability and carrying capacity. The novelty of the proposed DRL approach lies in its new characterization of system states, the embedment of heuristics-guided action choice, and the integration of rule-interposing into DRL training and implementation. The computational complexities of the heuristics and the overall

DQN training are investigated. The effectiveness of the approach is demonstrated through extensive numerical analysis. The results show the benefits brought by the heuristics-guided action choice, rule-interposing, and having time-related information in the state space in DRL training, the near-optimality of the solutions obtained, and the superiority of the proposed approach over existing methods in terms of solution quality, computation time, and scalability.

With its comprehensive and detailed specifications of states, actions, and rewards, the proposed approach not only has the potential to improve the efficiency of crowdshipping operation planning, but provides a new avenue that may be adapted to other pickup and delivery problems and vehicle routing contexts. For example, another type of crowdshipping with all requests originating from a central location (depot) can be viewed as a special case of the problems investigated in this paper. Also, while we consider dedicated crowdsourcees in the paper, the proposed DRL-based approach can be conveniently adapted to the context of opportunistic crowdsourcees given the origin and destination of the original trip of each crowdsourcee.

For possible extension of the proposed approach, we suggest a few directions. First, future efforts could be made to investigate a dynamic version of the problem. In this case, different initial states should be considered for different problem instances. Each time right before an assignment, the system state needs to reflect en-route crowdsourcees and idle crowdsourcees, the latter including those left unassigned from the previous assignment and new arrivals. Similarly, system state needs to encompass information of unassigned requests, including those left unassigned from the previous assignment and new arrivals. Second, in the real world the pickup and delivery locations of shipping requests are usually in different spatial distributions (e.g., the locations of restaurants/retail stores in a city may be quite different from the locations of residential buildings), which gives rise to the need for proactively relocating idle crowdsourcees to balance the spatial distribution of crowdsourcee supply and request pickup demand. It will be interesting to explore how to incorporate relocation decisions in the DRL framework. A third direction is to explore other DRL algorithms, including the effect of the state space dimension on training efficiency of those algorithms as compared to DQN. Lastly, some behavioral aspects, e.g., a crowdsourcee rejects an assigned request, could be added to further enrich the flexibility of the DRL model.

Acknowledgment

This research is funded by the National Science Foundation under Grant Number CMMI-1663411. The financial support of the National Science Foundation is gratefully acknowledged. An earlier version of the paper was presented at the INFORMS 2020 Annual Meeting. We also thank the

1023 three anonymous reviewers and Professor Qiang Meng, the Associate Editor, for their constructive
1024 feedback which has helped us significantly improve the paper.
1025

Appendix A: Proofs of Remarks 1-6

Proof of Remark 1. We look at the dimension of each component in the three-tuple of $\{S^l, S^r, S^c\}$. S^l specifies: 1) the coordinate of each node; 2) the coordinate of the successor node of a pickup node (if the request is assigned); 3) the coordinate of the predecessor node of a delivery node (if the request is assigned); and 4) the coordinate of the first node visited by a crowdsourcee. The number of nodes is $2|J| + |K|$. Thus, the dimension of S^l is $2\big((2|J| + |K|) + |J| + |J| + |K|\big) = 8|J| + 4|K|$,where the multiplication by 2 is because each coordinate contains longitude and latitude. For the second component, S^r specifies: 1) slack time of each request; 2) unused service time of each request; and 3) occupation time of each request. The dimension of S^r is $|J| + |J| + |J| = 3|J|$. For the third component, S^c specifies: 1) the routing duration for each crowdsourcee; 2) total delivery time violation of each crowdsourcee route; 3) remaining available time for each crowdsourcee; and 4) capacity violation of each crowdsourcee route. The dimension of S^c is $|K| + |K| + |K| + |K| = 4|K|$. So overall, the dimension of the state space is $8|J| + 4|K| + 3|J| + 4|K| = 11|J| + 8|K|$. ■

Proof of Remark 2. We first investigate the computational complexity of insertion for each of the three steps, based on which the overall computational complexity can be drawn. Step 1 requires calculation of slack time of at most $|J|$ requests, which will be directly extracted from the system state thus taking a constant time. Selecting the request with the smallest slack time requires sorting, whose complexity is $O(|J|\log|J|)$. So the overall complexity of Step 1 is $O(|J|\log|J|)$.

Step 2 requires calculation of the distance between the selected request and the end of each crowdsourcee routes. There are at most $|K|$ crowdsourcee routes. Thu, the distance calculation has a complexity of $O(|K|)$. Once the distances are obtained, a sorting is needed to identify the smallest distance, whose complexity is $O(|K|\log|K|)$. So the overall complexity of Step 2 is $O(|K|\log|K|)$.

Step 3 performs intra-route move. Given the limited number of requests a crowdsourcee can carry, the computation time for intra-route operation is bounded by a constant (see proof of Remark 3). The computation time for subsequent feasibility check is also bounded by a constant. The worst case is that we check feasibility of inserting the request to all crowdsourcee routes and finds none is feasible for the request. So the complexity is $O(|K|)$.

In this worst case, we need to move to the next request in the sorted list from Step 1, and perform Step 2 for the request. The overall worst case is that we check every request. Thus, the complexity of Steps 2-3 combined is $O(|J||K|\log|K|)$. Given that $|K| \leq |J|$ (i.e., the number of crowdsourcees is no

more than the number of requests) and the complexity of Step 1 is $O(|J|\log|J|)$, the overall complexity of insertion is $O(|J|^2\log|J|)$. ■

Proof of Remark 3. Similar to the proof of Remark 1, we first look into the computational complexity of each step in insertion. Step 1 requires sorting of at most $|J|$ crowdsourcee routes based on each route's remaining available time (which comes directly from the system state). Thus, the complexity of Step 1 is $O(|J|\log|J|)$. For Step 2, the computation time is bounded by a constant. This is because the number of requests that can be accommodated by a route is bounded given the limited carrying capacity of a crowdsourcee. Therefore, the number of possible moves in this step in a route is bounded. For Step 3, it involves sorting of the moves based on routing cost and comparison with the original routing cost. Again, given that the number of possible moves is bounded, the computation complexity of this step is a constant. Thus, overall, the computational complexity of intra-route move is $O(|J|\log|J|)$. ■

Proof of Remark 4. Step 1 of an inter-route move requires sorting of the assigned requests based on occupation time (which comes directly from the system state), thus having a computational complexity of $(|J|\log|J|)$. The computational complexity of Step 2 is $O(|K|\log|K|)$, as it follow the same step in insertion. Similar to the argument in intra-route move, the computation complexity of Step 3 is a constant as the number of possible moves is bounded (because the number of requests that can be accommodated by a route is bounded). Considering that $|J| \geq |K|$, the overall complexity is $O(|J|\log|J|)$. ■

Proof of Remark 5. Step 1 of a 1-exchange move involves sorting assigned requests based on unused service time (which comes from the system state), thus having a computation complexity of $O(|J|\log|J|)$. Step 2 does not involve further computation, as sorting is already done (excluding the route associated with the first selected request does not require another sorting). Step 3 exchanges the selected requests, which takes a constant time. Step 4 performs intra-route move of the two requests in their respective new routes, whose computational complexity is a constant following the same argument as in the proof of Remark 2. Thus, the overall complexity of 1-exchange is $O(|J|\log|J|)$. ■

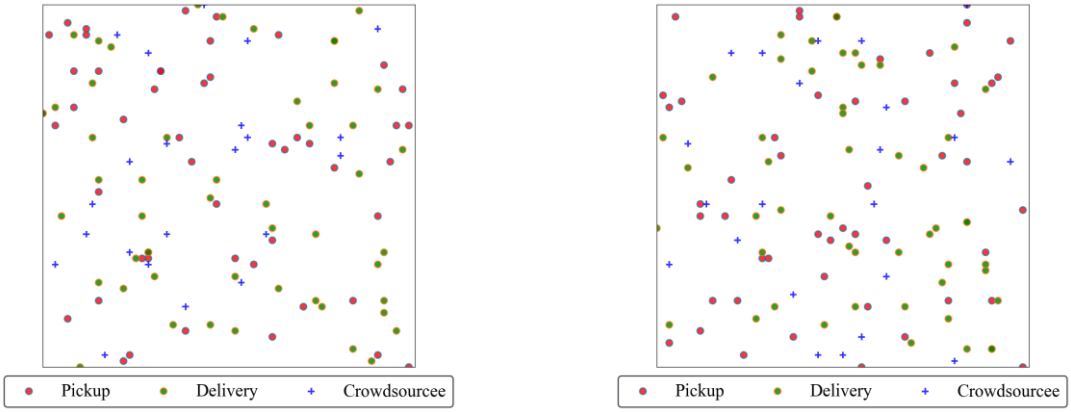
Proof of Remark 6. The complexity of DQN training depends on the number of parameters in the DNN to be trained. First, recall from Remark 1 that the state space has a dimension of $(11|J| + 8|K|)$.

Thus, the first layer has up to $(11|J| + 8|K| + 1)e$ parameters (since e is the upper bound on the number of neurons in a hidden layer). The subsequent layers each have up to $e^2 + e$ parameters. The DNN outputs the Q-values for each of the five action types, thus associated with up to $5(e + 1)$ parameters. Overall, the DNN has up to $(11|J| + 8|K| + 1)e + (\ell - 1)(e^2 + e) + 5(e + 1) = (11|J| + 8|K| + \ell + 5)e + (\ell - 1)e^2 + 5$ parameters to update in each time step. Note also that in each time step, a minibatch of $|M_{\text{sub}}|$ experiences are involved. In addition, a heuristic will be performed to execute a specific action for the chosen action type. Among the five action types, the greatest complexity occurs to insertion which has a complexity of $O(|J|^2 \log |J|)$ (see Remarks 2-5). Further recognizing that $|J| \geq |K|$, the complexity of one time step is $O([(|J| + \ell)e + \ell e^2]|M_{\text{sub}}| + |J|^2 \log |J|)$. Given that training takes I episodes each with up to T time steps, the overall complexity of DQN training is $O(\{[(|J| + \ell)e + \ell e^2]|M_{\text{sub}}| + |J|^2 \log |J|\} IT)$.

Two points are worth mentioning. First, we keep both $[(|J| + \ell)e + \ell e^2]|M_{\text{sub}}|$ and $|J|^2 \log |J|$ terms in the complexity expression, as it is not clear *a priori* which of the two terms dominates the computation time. Second, the complexity expression is parameterized by the number of episodes I, which typically cannot be determined before the training is carried out as it depends on the characteristics of the instances used for learning. ■

Appendix B: Illustration of problem instances used in training and testing

We randomly pick two instances, one from training and one from testing, to display the pickup and delivery locations of requests and origins of crowdsourcees. Overall, no significant differences (apart from the effect due to randomness) among the instances are discerned.



Training instance Testing instance

Fig. B1. Illustration of two randomly picked problem instances from training and testing

Appendix C: Identification of the specific action to take given the action type under a DRL algorithm without heuristics-guided action choice

Insertion	
Step 1:	*Select a request.* Among the unassigned requests, randomly select an unassigned request.
Step 2:	*Insert the request to a route.* Insert the request to the end of a randomly picked crowdsourcee route (which can be an existing or a new route). If the insertion is not feasible, then randomly pick another crowdsourcee route. If a feasible insertion cannot be found, then do nothing.
Intra-route move	
Step 1:	*Select a route.* Select the crowdsourcee route with the largest remaining available time (based on Rule 1 in subsection 3.4.1).
Step 2:	*Move a request from the route to a different location on the same route.* Randomly pick a request from the route. Enumerate all feasible moves of the pickup and delivery nodes of the request on the route. Pick the move with the maximum reduced cost. If such a move does not exist, then randomly pick another request and do the same. If such a move cannot be found after enumerating all requests on the route, then do nothing.
Inter-route move	
Step 1:	*Select a request.* Select the crowdsourcee route with the largest occupation time (based on Rule 1 in subsection 3.4.1).
Step 2:	*Move the request to the end of a different route.* Randomly select a request from the route. Investigate moving the request to the end of a different route that is also randomly picked. If the move if feasible, perform the move. Otherwise, randomly pick another route and investigate moving the request to the end of the route. If the request cannot be moved to the end of any different route, then do nothing.
1-exchange	
Step 1:	*Select two routes.* Select the two crowdsourcee routes with the largest and the second largest unused service time (based on Rule 1 in subsection 3.4.1).
Step 2:	*Select requests from the two routes and exchange.* Randomly select a request from each routes and exchange their locations.

Note that for intra-route move, inter-route move, and 1-exchange, we do not consider Rule 2 of subsection 3.4.2 since the rule is related to heuristics-guided action choice.

Appendix D: MILP formulation of the crowdshipping problem

The crowdshipping problem is a pickup and delivery problem with time and capacity constraints. We consider the following MILP model which is based on Hamiltonian tour formulation (Lu and Dessouky, 2004). In the formulation, request nodes are ordered such that the first $|J|$ nodes are pickup nodes, and the remaining nodes are delivery nodes which follow the same order as their associated pickup nodes. Recall that in the paper the set of request nodes is J. Then we use $J^+ = \{1, 2, \dots, |J|\}$ to denote the set of pickup nodes, and $J^- = \{|J| + 1, |J| + 2, \dots, 2|J|\}$ to denote the set of delivery nodes. We further introduce set $\mathbb{N} = J \cup K = J^+ \cup J^- \cup K$, which contains in sequence nodes in J^+, nodes in J^-, and crowdsourcee origin nodes in $K = \{2|J| + 1, 2|J| + 2, \dots, 2|J| + |K|\}$.

Among the parameters, q_j denotes the weight of request at node $j \in J$. $q_j > 0$ if j is a pickup node, and $q_j < 0$ if j is a delivery node. c_{ij} and t_{ij} denote respectively the cost and time while a crowdsourcee traverses link (i, j). Because crowdsourcee routes are constructed sequentially based on Hamiltonian tour formulation, $c_{ij} = 0$ if $i \in J$ and $j \in K$. $\mathcal{C}$ denotes carrying capacity of a crowdsourcee.

The MILP model has four set of decision variables: 1) $\mathbf{x} = \{x_{ij}; i, j \in \mathbb{N}, i \neq j\}$, which are binary indicating whether node i is right before node j in the Hamiltonian tour; 2) $\mathbf{y} = \{y_{ij}; i, j \in \mathbb{N}, i \neq j\}$, which are also binary indicating whether node i is before node j in the Hamiltonian tour; 3) $\mathbf{Q} = \{Q_i; i \in J\}$, which are continuous variables deciding the carrying load of a crowdsourcee right after visiting a request node i; and 4) $\mathbf{T} = \{T_i; i \in J\}$, which are continuous variables deciding the departure time of a crowdsourcee from a request node i.

$$\min_{\mathbf{x},\mathbf{y},\mathbf{Q},\mathbf{T}} \sum_{i\in\mathbb{N}}\sum_{j\in J} c_{ij}x_{ij} \quad \text{(G1)}$$

s.t.

Routing sequence constraints

$$\sum_{i\in\mathbb{N}} x_{ij} = 1 \qquad \forall j \in \mathbb{N} \quad \text{(G2)}$$

$$\sum_{j\in\mathbb{N}} x_{ij} = 1 \qquad \forall i \in \mathbb{N} \quad \text{(G3)}$$

$$y_{ki} \le y_{kj} + (1 - x_{ij}) \qquad \forall i,j,k \in \mathbb{N} \text{ and } j \ne 2|J|+1 \quad \text{(G4)}$$

$$y_{ki} \ge y_{kj} + (x_{ij} - 1) \qquad \forall i,j,k \in \mathbb{N} \text{ and } j \ne 2|J|+1 \quad \text{(G5)}$$

$$x_{ij} \le y_{ij} \qquad \forall i,j \in \mathbb{N} \quad \text{(G7)}$$

$$y_{i,|J|+i} = 1 \qquad \forall i \in J^+ \quad \text{(G8)}$$

$$y_{|J|+i,i} = 0 \qquad \forall i \in J^+ \quad \text{(G9)}$$

$$y_{ij} = y_{|J|+i,j} \qquad \forall i \in J^+, j \in K \quad \text{(G10)}$$

$$y_{ij} = 1 \qquad \forall i,j \in K \text{ and } i < j \quad \text{(G11)}$$

$$y_{ij} = 0 \qquad \forall i,j \in K \text{ and } i > j \quad \text{(G12)}$$

Capacity constraints

$$Q_i + q_j - Q_j \le \mathcal{M}(1 - x_{ij}) \qquad \forall i,j \in \mathbb{N} \quad \text{(G13)}$$

$$Q_i + q_j - Q_j \ge \mathcal{M}(x_{ij} - 1) \qquad \forall i,j \in \mathbb{N} \quad \text{(G14)}$$

$$Q_j \le \mathcal{C} \qquad \forall j \in J \quad \text{(G15)}$$

Delivery time window and crowdsourcee time availability constraints

$$T_i + t_{ij} + s - T_j \le \mathcal{M}(1 - x_{ij}) \qquad \forall i \in \mathbb{N}, j \in J \quad \text{(G16)}$$

$$T_i + t_{ij} + s - T_j \ge \mathcal{M}(x_{ij} - 1) \qquad \forall i \in \mathbb{N}, j \in J \quad \text{(G17)}$$

$$T_j \le t^l_{d_j} \qquad \forall i \in \mathbb{N}, j \in J^- \quad \text{(G18)}$$

$$T_j \ge t^e_{p_j} \qquad \forall i \in \mathbb{N}, j \in J^+ \quad \text{(G19)}$$

$$T_{|J|+i} \ge T_i \qquad \forall i \in J^+ \quad \text{(G20)}$$

$$T_i - t^k_{\text{end}} \le \mathcal{M}(1 - x_{i,k+1}) \qquad \forall i \in J^-, k \in K \setminus \{2|J| + |K|\} \quad \text{(G21)}$$

$$T_i - t^{2|J|+|K|}_{\text{end}} \le \mathcal{M}(1 - x_{i,2|J|+1}) \qquad \forall i \in J^- \quad \text{(G22)}$$

$$T_i \ge t^i_{\text{start}} \qquad \forall i \in K \quad \text{(G23)}$$

Integrality, non-negativity, and fixed-value constraints

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in N \quad \text{(G24)}$$

$$y_{ij} \in \{0,1\} \qquad \forall i,j \in N \quad \text{(G25)}$$

$$Q_i \ge 0 \qquad \forall i \in J \quad \text{(G26)}$$

$$Q_i = 0 \qquad \forall i \in K \quad \text{(G27)}$$

$$T_i \ge 0 \qquad \forall i \in J \quad \text{(G28)}$$

The objective function (G1) minimizes total routing cost. The constraints are organized in four groups. The first group relates to routing sequence. Constraints (G2)-(G3) stipulate that each node is visited exactly once in the Hamiltonian tour. Constraints (G4)-(G5) ensure that the precedence relationship of a node k with respect to two connected nodes (i and j with $x_{ij} = 1$) should be consistent, i.e., $y_{ki} = y_{kj}$. Since the origin of crowdsourcee 1 is the start of the Hamiltonian tour, it is

meaningless to talk about its preceding node (that is why $j \neq 2|J|+1$). The case of $(i,j) = (2|J|+|K|, 2|J|+1)$ is excluded since the end of crowdsourcee $|K|$'s route will be artificially connected to crowdsourcee 1's origin (since it is a Hamiltonian tour). (G7) says that the case of immediate precedence is more restricted than general precedence. (G8)-(G9) specifies that for a request, the pickup node must be visited before the delivery node. (G10) specifies that a crowdsourcee's origin cannot be in between the pickup and delivery nodes of a request. Constraints (G11)-(G12) the precedence relationship between two crowdsourcees' origins follow their orders in K.

For the second group, capacity-related constraints, (G13)-(G14) updates the carrying load by a crowdsourcee for two consecutively visited nodes. (G15) constrains that the carrying load does not exceed the carrying capacity of a crowdsourcee. For the third group, time-related constraints, (G16)-(G17) calculates the departure time from a node j based on the departure time from its immediate preceding node i, travel time from i to j, and stopping time at j. (G18) says that the actual delivery time (arrival time) at a delivery node j should be no later than the latest delivery time. Similarly, (G19) says that the actual pickup time at a pickup node j should be no earlier than the earliest pickup time. (G20) means that the time of visiting a delivery node should be no earlier than the time of visiting the corresponding pickup node. (G21)-(G22) stipulate that a crowdsourcee route needs to end earlier than the end of the crowdsourcee's available time. (G22) is written separately for the last crowdsourcee $|K|$ because, based on the Hamiltonian tour formulation, the end of crowdsourcee $|K|$'s route connects back to crowdsourcee 1's origin. Furthermore, the leaving time from the origin should be no earlier than the start of the available time of a crowdsourcee (constraint (G23)). With these constraints, the feasibility of crowdsourcee routes as presented in Definition 1 is ensured. The final group of constraints specifies the integrality, non-negativity, and fixed-value constraints of the decision variables. In particular, at the origin, a crowdsourcee does not carry loads (constraint (G27)).

References

1. Ahamed, T., Zou, B., 2020. Multi-tier adaptive memory programming and cluster- and job-based relocation for distributed on-demand crowdshipping. *Working Paper*. Department of Civil, Materials, and Environmental Engineering, University of Illinois at Chicago.
2. Al-Abbasi, A.O., Ghosh, A. and Aggarwal, V., 2019. Deeppool: Distributed model-free algorithm for ride-sharing using deep reinforcement learning. IEEE Transactions on Intelligent Transportation Systems, 20(12), pp.4714-4727.
3. Arslan, A., Agatz, N. and Klapp, M., 2020. Operational Strategies for On-demand Personal Shopper Services.
4. Braekers, K. and Kovacs, A.A., 2016. A multi-period dial-a-ride problem with driver consistency. Transportation Research Part B: Methodological, 94, pp.355-377.

5. Bello, I., Pham, H., Le, Q.V., Norouzi, M. and Bengio, S., 2016. Neural combinatorial optimization with reinforcement learning. arXiv preprint arXiv:1611.09940.
6. Berbeglia, G., Cordeau, J.F. and Laporte, G., 2010. Dynamic pickup and delivery problems. European journal of operational research, 202(1), pp.8-15.
7. Chen, X., Ulmer, M.W. and Thomas, B.W., 2019. Deep Q-Learning for Same-Day Delivery with a Heterogeneous Fleet of Vehicles and Drones. arXiv preprint arXiv:1910.11901.
8. Dai, H., Khalil, E., Zhang, Y., Dilkina, B. and Song, L., 2017. Learning combinatorial optimization algorithms over graphs. In Advances in Neural Information Processing Systems (pp. 6348-6358).
9. Ghilas, V., Demir, E. and Van Woensel, T., 2016. A scenario-based planning for the pickup and delivery problem with time windows, scheduled lines and stochastic demands. Transportation Research Part B: Methodological, 91, pp.34-51.
10. Kafle, N., Zou, B. and Lin, J., 2017. Design and modeling of a crowdsource-enabled system for urban parcel relay and delivery. Transportation research part B: methodological, 99, pp.62-82.
11. Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
12. Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P., 1983. "Optimization by Simulated Annealing," Science vol. 220, no. 4598, pp. 671-680.
13. Kool, W., Van Hoof, H. and Welling, M., 2018. Attention, learn to solve routing problems!. arXiv preprint arXiv:1803.08475.
14. Le, T.V., Stathopoulos, A., Van Woensel, T. and Ukkusuri, S.V., 2019. Supply, demand, operations, and management of crowd-shipping services: A review and empirical evidence. Transportation Research Part C: Emerging Technologies, 103, pp.83-103.
15. Liu, M., Luo, Z. and Lim, A., 2015. A branch-and-cut algorithm for a realistic dial-a-ride problem. Transportation Research Part B: Methodological, 81, pp.267-288.
16. Lu, Q. and Dessouky, M.M., 2006. A new insertion-based construction heuristic for solving the pickup and delivery problem with time windows. European Journal of Operational Research, 175(2), pp.672-687.
17. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. nature, 518(7540), pp.529-533.
18. Nanry, W.P. and Barnes, J.W., 2000. Solving the pickup and delivery problem with time windows using reactive tabu search. Transportation Research Part B: Methodological, 34(2), pp.107-121.
19. Nazari, M., Oroojlooy, A., Snyder, L. and Takác, M., 2018. Reinforcement learning for solving the vehicle routing problem. In Advances in Neural Information Processing Systems (pp. 9839-9849).
20. Oda, T. and Joe-Wong, C., 2018, April. MOVI: A model-free approach to dynamic fleet management. In IEEE INFOCOM 2018-IEEE Conference on Computer Communications (pp. 2708-2716). IEEE.
21. Shao, S., Xu, S.X. and Huang, G.Q., 2020. Variable neighborhood search and tabu search for auction-based waste collection synchronization. Transportation Research Part B: Methodological, 133, pp.1-20.
22. Singh, A., Al-Abbasi, A. and Aggarwal, V., 2019, December. A reinforcement learning based algorithm for multi-hop ride-sharing: Model-free approach. In Neural Information Processing Systems (Neurips) Workshop.
23. Sutton, R.S. and Barto, A.G., 2018. Reinforcement learning: An introduction. MIT press.
24. Tan, K.C., Lee, L.H., Zhu, Q.L. and Ou, K., 2001. Heuristic methods for vehicle routing problem with time windows. Artificial intelligence in Engineering, 15(3), pp.281-295.
25. Van Laarhoven, P.J. and Aarts, E.H., 1987. Simulated annealing. In Simulated annealing: Theory and applications (pp. 7-15). Springer, Dordrecht.
26. Wang, Y., Zhang, D., Liu, Q., Shen, F. and Lee, L.H., 2016. Towards enhancing the last-mile delivery: An effective crowd-tasking model with scalable solutions. Transportation Research Part E: Logistics and Transportation Review, 93, pp.279-293.

27. Watkins, C.J. and Dayan, P., 1992. Q-learning. Machine learning, 8(3-4), pp.279-292.
28. Yu, J., Yu, W. and Gu, J., 2019. Online vehicle routing with neural combinatorial optimization and deep reinforcement learning. IEEE Transactions on Intelligent Transportation Systems, 20(10), pp.3806-3817.

Tanvir Ahamed: Conceptualization, Methodology, Coding, Investigation, Writing – Original Draft, Writing – Review & Editing, Visualization. **Bo Zou:** Conceptualization, Methodology, Investigation, Writing – Original Draft, Writing – Review & Editing, Supervision. **Nahid Parvez Farazi:** Methodology, Investigation, Writing – Original Draft, Writing – Review & Editing. **Theja Tulabandhula:** Methodology, Investigation, Writing – Review & Editing.