Future Generation Computer Systems 128 (2022) 333–348

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Offloading dependent tasks in multi-access edge computing: A multi-objective reinforcement learning approach

Fuhong Song, Huanlai Xing *, Xinhan Wang, Shouxi Luo, Penglin Dai, Ke Li

School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu 610031, PR China

ARTICLE INFO

Article history:
Received 29 June 2021
Received in revised form 6 September 2021
Accepted 9 October 2021
Available online 22 October 2021

Keywords:
Computation offloading
Dynamic preferences
Multi-access edge computing
Multi-objective reinforcement learning
Task dependency

ABSTRACT

This paper studies the problem of offloading an application consisting of dependent tasks in multi-access edge computing (MEC). This problem is challenging because multiple conflicting objectives exist, e.g., the completion time, energy consumption, and computation overhead should be optimized simultaneously. Recently, some reinforcement learning (RL) based methods have been proposed to address the problem. However, these methods, called single-objective RLs (SORLs), define the user utility as a linear scalarization. The conflict between objectives has been ignored. This paper formulates a multi-objective optimization problem to simultaneously minimize the application completion time, energy consumption of the mobile device, and usage charge for edge computing, subject to dependency constraints. Moreover, the relative importance (preferences) between the objectives may change over time in MEC, making it quite challenging for traditional SORLs to handle. To overcome this, we first model a multi-objective Markov decision process, where the scalar reward is extended to a vector-valued reward. Each element in the reward corresponds to one of the objectives. Then, we propose an improved multi-objective reinforcement learning (MORL) algorithm, where a tournament selection scheme is designed to select important preferences to effectively maintain previously learned policies. The simulation results demonstrate that the proposed algorithm obtains a good tradeoff between three objectives and has significant performance improvement compared with a number of existing algorithms.

1. Introduction

Recent years have witnessed the rapid advance of communication technologies and mobile devices (MDs), and various mobile applications are emerging, such as face recognition, virtual/augmented reality (VR/AR), and mobile healthcare [1,2]. These applications result in a significant surge in demand for computing and storage resources. However, MDs may not support these computing-intensive applications due to their limited computing resources and battery capacity [3]. The tension between computing-intensive applications and resource-constrained MDs creates a bottleneck for achieving satisfactory quality of experience (QoE). Multi-access edge computing (MEC) [4,5] is a promising solution to overcome the hurdle by providing abundant computing and storage resources around MDs. Computation offloading in MEC is a process that migrates computing-intensive applications from resource-constrained MDs to resource-rich edge servers [6]. This process not only alleviates the computation pressure on MDs, but also guarantees the QoE demands for mobile applications [7].

In real-world scenarios, many applications, such as VR/AR and face recognition, consist of multiple tasks, making fine-grained computation offloading possible [6,8]. We consider the computation offloading of a single application in MEC. When offloading technology is adopted, one of the most significant issues is the computation partitioning problem. Computation partitioning decides which tasks in an application to be offloaded to edge servers and which ones to be performed locally on MDs, such that the QoE of end-users is satisfied. Nevertheless, the dependencies between tasks cannot be neglected because the results obtained by executing some tasks are prerequisites for others. When offloading such an application, if we do not take the dependencies between tasks into account, this application may not be executed successfully. Thus, offloading dependent tasks is important in MEC. Most studies on dependent task offloading are based on heuristic or metaheuristic algorithms [8–11], due to the NP-hardness of the problem. However, these studies fail to fully adapt to dynamic MEC scenarios as the aforementioned algorithms are usually of high complexity and thus time-consuming.

Deep reinforcement learning (DRL) [12,13], which combines reinforcement learning (RL) with deep neural network (DNN) [14], can deal with sequential decision-making problems in dynamic MEC scenarios. The reason is that DRL can adapt their behavior

* Corresponding author.
E-mail address: hxx@home.swjtu.edu.cn (H. Xing).

https://doi.org/10.1016/j.future.2021.10.013
0167-739X/

to the changes of MEC scenarios, such as time-varying wireless channel qualities. The problem of offloading dependent tasks (ODT) belongs to the sequential decision-making problem that DRL can address. In addition, an MD needs to obtain an offloading solution quickly because many applications, such as VR/AR, have short deadlines. DRL can generate a solution quite quickly once its training is over, thus satisfying the demands of delay-sensitive applications. This is our motivation to develop an RL-based algorithm to address the ODT problem in a dynamic MEC system.

Recently, many researchers have adopted RL-based methods [15,16] to deal with the uncertainties of MEC systems and achieve promising performance for computation offloading. The application completion time, energy consumption of the MD, and usage charge for edge computing are usually considered to be the three most important criteria for performance evaluation. The three objectives conflict with each other, meaning that improving one of them would deteriorate the others. However, recent researches mostly defined the user utility in computation offloading as a linear scalarization with weights per objective, i.e., the weighted sum method. The fact that the objectives conflict with each other has been ignored, thus resulting in poor computation offloading performance.

On the other hand, the weights per objective were assumed unchanged during learning and execution. So, the problem of offloading dependent tasks in MEC was addressed using single-objective RLs (SORLs). It is worth noting that the weights represent the relative importance (i.e., preferences) between objectives. Although this suffices in cases where the preferences are known in advance, the learning policy is limited when adapted to scenarios with different preferences. This is because, in most cases, the preferences are not known beforehand and may change over time, especially in dynamic MEC systems. For example, end-users may focus on the completion time of delay-sensitive applications. However, when the computing budgets of end-users are insufficient, they would like to decrease the usage charge. When MDs are of low battery levels, these end-users prefer to reduce energy consumption. The dynamic preferences between objectives further increase the complexity of MEC systems, and bringing a big challenge to offloading making-decision. Thus, SORLs are not suitable for MEC scenarios with dynamic preferences. This motivates us to adapt a multi-objective reinforcement learning (MORL) algorithm to the problem concerned in the paper.

MORLs have attracted increasingly more research attention [17–19]. Unlike SORL with a scalar reward, MORL has a vector-valued reward in which each element corresponds to a specific objective. Through learning weight-dependent multi-objective Q-value vectors, MORL with dynamic weight setting (MORL-DWS) [19] has great potential to handle scenarios with ever-changing user preferences. This algorithm has been successfully applied to various domains, such as energy, health, transportation, and dialog system [19,20].

In this paper, we define a multi-objective optimization problem of offloading dependent tasks in dynamic MEC systems. An improved MORL-DWS algorithm is proposed to solve it. The main contributions are summarized as follows.

- We formulate the problem of offloading dependent tasks (ODT) in MEC, with the application completion time (ACT), energy consumption (EC) of the MD, and usage charge (UC) for edge computing minimized simultaneously. This problem is quite challenging because three objectives conflict with each other, such as offloading can reduce energy consumption while increasing usage charge.
- To solve the ODT problem, we model a multi-objective Markov decision process (MOMDP) with a vector-valued reward. Based on the MOMDP, we propose an improved MORL-DWS with a tournament selection (TS) scheme, called MORL-ODT. The TS scheme selects significant preferences from the set of encountered preferences to effectively maintain previously learned policies. To our knowledge, it is the first work to apply MORL to the computation offloading problem.
- We conduct extensive simulation experiments using a set of generated test instances with various topologies and task profiles. The simulation results show that the proposed algorithm can obtain promising tradeoffs between the three objectives in most cases and outperforms four state-of-the-art RLs and one metaheuristic against several evaluation criteria, including the regret, adaptation error, completion time, energy consumption, usage charge, and network usage.

The rest of the paper is organized as follows. The related work is reviewed in Section 2. In Section 3, we describe the system model and problem formulation. Section 4 briefly reviews the MOMDP and original MORL-DWS. In Section 5, we introduce the proposed MORL-ODT in detail. Simulation results are presented in Section 6. Finally, Section 7 concludes the paper.

2. Related work

There have been a large number of methods for offloading dependent tasks. We classify them into two categories.

2.1. Heuristic and metaheuristic based methods

Sundar et al. [8] proposed a heuristic algorithm to achieve an efficient solution, aiming at minimizing the execution cost of applications under the application completion time constraint. Zhao et al. [9] modeled the ODT problem with service caching in a homogeneous MEC scenario and presented a favorite successor based heuristic algorithm to minimize the application completion time. Zhang et al. [21] modeled the application execution as a delay-constrained workflow scheduling problem. Considering whether execution restrictions existed in applications, the authors proposed a one-climb policy and Lagrange relaxation algorithm to minimize the energy consumption of MDs. Maray et al. [22] proposed a heuristic algorithm to reduce the completion time of dependent tasks, subject to application deadline constraints. Mahmoodi et al. [23] introduced a wireless aware scheduling and computation offloading algorithm to shorten the execution time through processing appropriate tasks of an application in parallel. Sahni et al. [24] studied the problem of task offloading and network flow scheduling. A joint task offloading and flow scheduling heuristic algorithm was developed to reduce the average completion time of tasks. Peng et al. [10] designed a task scheduling method based on the whale optimization algorithm (WOA) to simultaneously minimize the application completion time and energy consumption. Habob et al. [11] presented a genetic-algorithm (GA) based solution to minimize the offloading latency and failure probability. Huang et al. [25] took the security, energy consumption, and application completion time into account. In their study, a GA was adopted to minimize the energy consumption of MDs under application deadline constraints. Song et al. [26] studied the multi-objective computation offloading problem. The application completion time and energy computation of MDs were minimized by a multi-objective evolutionary algorithm based on decomposition (MOEA/D). Xie et al. [27] proposed a directional and non-local-convergent particle swarm optimization (PSO) algorithm to optimize the completion time and cost of applications.

Table 1
A side-by-side comparison of heuristic and metaheuristic based offloading methods.

Ref.	Utilized technique	Performance metrics	Date set	Evaluation tools	Advantages	Disadvantages
[8]	Heuristic	• Completion time • Cost	• Random	• Simulation (NA)	• Fast convergence	• High complexity • Poor adaptation
[9]	Heuristic	• Completion time	• Random	• Simulation (NA)	• Effective service caching	• Single objective • Poor adaptation
[21]	Heuristic	• Energy consumption	• Random	• Simulation (NA)	• Reasonable complexity	• Single objective • Hard to implement
[22]	Heuristic	• Completion time	• Real-world	• Simulation (NS3)	• Low complexity	• Single objective
[23]	Heuristic	• Energy consumption	• Real-world	• Simulation (NA)	• Good scalability • Multiple constraints	• Single objective • Time-consuming
[24]	Heuristic	• Completion time • Running time	• Random	• Simulation (NA)	• Consider network flow scheduling	• Single objective • Poor scalability
[10]	WOA	• Completion time • Energy consumption	• Random	• Simulation (Matlab)	• Provide a tradeoff between objectives	• Do not consider time-varying channel
[11]	GA	• Completion time • Failure probability	• Random	• Simulation (NA)	• Less offloading failure probability	• High complexity • Poor convergence
[25]	GA	• Energy consumption	• Random	• Simulation (Python)	• Consider security	• Single objective • Single access point
[26]	MOEA/D	• Completion time • Energy consumption	• Random	• Simulation (Python)	• Provide a tradeoff between objectives	• Poor scalability
[27]	PSO	• Completion time • Cost	• Random • Real-world	• Simulation (WorkflowSim)	• Provide a tradeoff between objectives	• High complexity • Poor convergence

A side-by-side comparison of heuristic and metaheuristic based offloading methods with respect to the utilized technique, performance metrics, data set, evaluation tools, advantages, and disadvantages is provided in Table 1. Heuristic-based methods have been regarded as a good choice for handling static MEC scenarios. However, they are hardly adapted to a dynamic MEC system. This is because heuristics need to be re-executed whenever the MEC environment changes. Although metaheuristic-based methods are more suitable for addressing dynamic MEC environments and obtain better solutions than heuristics-based ones, they often suffer from rapid diversity loss and premature convergence. Thus, there is no guarantee for metaheuristics to always achieve excellent computation offloading performance.

2.2. Reinforcement learning based methods

Reinforcement learning emphasizes how to select actions to maximize the expected returns by interacting with the environment [28]. Yan et al. [15] proposed an actor-critic framework based RL to minimize the completion time and energy consumption. An actor network was employed to learn optimal policies. A critic network evaluated the performance of each policy. Wang et al. [16] put forward an RL-based offloading framework to minimize the application completion time. A sequence-to-sequence neural network was adopted to represent offloading policies. In a follow-up study [29], the authors presented a task offloading algorithm based on meta RL to optimize the application completion time. However, their studies did not consider the energy consumption, one of the most important criteria for QoE evaluation. Wu et al. [30] proposed an adaptive task scheduling algorithm using the policy gradient-based REINFORCE to minimize the application completion time. Zhu et al. [31] applied a deep Q-learning to reduce the application completion time and energy consumption of MDs, simultaneously. Tang et al. [32] proposed an RL-based multi-job dependent task offloading algorithm in which a graph convolutional network was devised to extract the dependency information between tasks, aiming at minimizing the transmission and computation costs. Wang et al. [33] designed a temporal fusion pointer network-based RL (TFP-RL) to address the multi-objective workflow scheduling problem. To train the TFP-RL model, the authors used an asynchronous advantage actor-critic method. Shahidinejad et al. [34] adopted the long short-term memory model to evaluate the future Internet of Things (IoT) requests and developed an RL-based technique to make proper scaling decisions. Jazayeri et al. [35] proposed an RL-based algorithm to obtain the offloading decisions of modules in applications, striking a balance between power consumption and execution time. In a follow-up study, the authors [36] put forward a hidden Markov model-based method to obtain the best offloading destination to reduce the completion time and energy consumption of applications. Qu et al. [37] modeled a dynamic computation offloading problem for sequential dependent tasks in edge/cloud computing as a multi-objective optimization problem. A meta RL-based offloading framework was proposed to optimize the application completion time and energy consumption. Lu et al. [38] considered multi-service nodes and dependent tasks in heterogeneous MEC scenarios. A task offloading algorithm based on RL was proposed to reduce the latency, cost, and energy consumption of the MEC platform.

A side-by-side comparison of reinforcement learning based offloading methods is provided in Table 2. All the existing RLs are SORLs, and their aim is to optimize a scalar reward. If these SORLs are applied to the ODT problem concerned in this paper, user utility is likely to be defined as a linear scalarization, i.e., the weighted sum of all objectives. Thus, SORLs are limited to the case where the weights per objective (i.e., preferences) are known in advance and fixed during learning and execution. In other words, an SORL only learns optimal policy over a single preference after a run. In a dynamic MEC system, however, the preferences cannot be determined in advance and may change over time. In this case, an SORL requires re-learning on new preferences, often consuming a considerable amount of time for training. Unlike SORLs, MORL-DWS can generalize across preference changes by learning weight-dependent multi-objective Q-value vectors. This motivates us to adapt MORL-DWS to the ODT problem.

Table 3 clearly shows the differences between similar works and ours in terms of the optimization objective and reinforcement learning. It can be seen that we optimize the ACT, EC and UC, simultaneously, while most works only minimize one or two of them. Moreover, unlike existing works use SORLs, we adopt an MORL (i.e., MORL-DWS) to address the ODT problem.

3. System model and problem formulation

As shown in Fig. 1, we consider an MEC system consisting of one MD and a set of base stations. Each base station is equipped with an edge server. Let $\mathcal{M} = \{1, \ldots, M\}$ denote the set of edge servers in the MEC system, where M is the number of edge

Table 2
A side-by-side comparison of reinforcement learning based offloading methods.

Ref.	Utilized technique	Performance metrics	Date set	Evaluation tools	Advantages	Disadvantages
[15]	RL	• Completion time • Energy consumption	• Random	• Simulation (NA)	• Consider resource allocation	• Poor scalability
[16]	RL	• Completion time	• Random	• Simulation (Python)	• Low complexity	• Single objective
[29]	RL	• Completion time	• Random	• Simulation (Python)	• Low complexity • High efficiency	• Single objective
[30]	RL	• Completion time • Running time	• Random	• Simulation (Python)	• Fast convergence	• Single objective • High complexity
[31]	RL	• Completion time • Energy consumption	• Random	• Simulation (Python)	• Good convergence	• Do not consider usage charge
[32]	RL	• Cost	• Real-world	• Simulation (NA)	• Good scalability	• Do not consider Completion time
[33]	RL	• Completion time • Cost	• Random	• Simulation (NA)	• Adopt multi-agent	• Do not consider energy consumption
[34]	RL	• Latency • CPU utilization • Energy consumption	• Random • Real-world	• Simulation (NA)	• High accuracy	• Poor scalability
[35]	RL	• Completion time • Power consumption	• Real-world	• Simulation (iFogsim)	• Appropriate system model • Fast convergence	• Do not consider usage charge
[36]	RL	• Completion time • Power consumption • Network usage	• Real-world	• Simulation (iFogsim)	• Appropriate system model • Fast convergence	• Do not consider usage charge
[37]	RL	• Completion time • Energy consumption	• Random	• Simulation (NA)	• Good portability • Fast convergence	• Poor scalability
[38]	RL	• Completion time • Energy consumption	• Random • Real-world	• Simulation (iFogsim)	• Good scalability	• Poor convergence

Table 3
Differences between similar works and ours.

Ref.		[15]	[16]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[37]	[38]	Ours
Optimization objective	ACT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	EC	✓				✓			✓	✓	✓	✓	✓	✓
	UC							✓					✓	✓
Reinforcement learning	SORL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
	MORL													✓

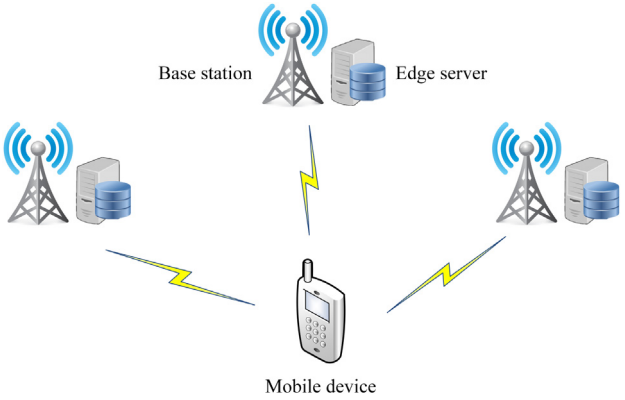

Fig. 1. MEC system.

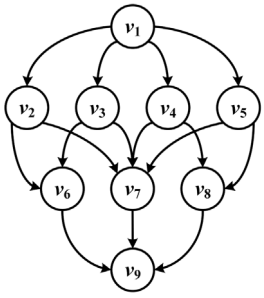

Fig. 2. Example DAG of an application.

servers. The MD has a computing-intensive application with a number of dependent tasks to run. As depicted in Fig. 2, we model the application as a directed acyclic graph (DAG) $G = (V, E)$, where the vertex and edge sets, V and E, represent the tasks and dependency constraints between them, respectively. Let $N = |V|$ denote the number of tasks in G. Let $pre(v_i)$ and $suc(v_i)$ be the sets of the immediate predecessors and successors of v_i, respectively. Edge $e(v_i, v_j)$ denotes the dependency constraint from tasks v_i to v_j ($i \neq j$), where $v_i \in pre(v_j)$ is an immediate predecessor of v_j, and $v_j \in suc(v_i)$ is an immediate successor of v_i. Dependency constraint $e(v_i, v_j)$ means task v_j cannot be launched until task v_i is completed. In a DAG, the task without any predecessor is referred to as the *start* task, and a task without any successor is referred to as an *end* task. For example, tasks v_1 and v_9 in Fig. 2 are the start and end tasks, respectively. Let $end(G)$ be the set of end tasks in G.

Each task in G is modeled as a tuple $v_i = \langle c_i, d_i, q_i \rangle$, where c_i is the number of CPU cycles required to execute v_i, and d_i and q_i are the input and output data sizes, respectively. The input data of v_i consist of the program code, input parameters, and output data generated by all its immediate predecessors in $pre(v_i)$. In computation offloading, each task in G can either be executed locally on the MD or offloaded to one of the M edge servers. Let $X = (x_1, \ldots, x_N)$ denote the execution location vector of all tasks in G, where $x_i \in \{0, 1, \ldots, M\}$ is the execution location of task v_i. If $x_i = 0$, v_i is executed on the MD, and otherwise v_i is offloaded to $x_i \in \mathcal{M}$. The main notations used in this paper are summarized in Table 4.

Table 4
Summary of main notations.

Notation	Definition
Notation used in system model	
c_i	Number of CPU cycles required to execute task v_i
$C(G)$	Usage charge for edge computing
d_i	Input data size of task v_i
$e(v_i, v_j)$	Dependency constraint from tasks v_i to v_j
$end(G)$	Set of end tasks in G
E	Set of the dependency constraints between tasks
$E(G)$	Energy consumption of the MD
f^l	Computing capability of the MD
f^s_m	Computing capability of the mth edge server
G	Application to be executed on the MD
$\mathcal{M}$	Set of edge servers
M	Number of edge servers
N	Number of tasks in application G
O	Execution order vector of all tasks in application G
$pre(v_i)$	Set of the immediate predecessors of task v_i
p^r	Receiving power of the MD
p^t	Transmission power of the MD
q_i	Output data size of task v_i
$suc(v_i)$	Set of the immediate successors of task v_i
$T(G)$	Application completion time
u_t	The tth task to be executed in execution order vector O
v_i	The ith task in application G
V	Set of the tasks in application G
x_i	Execution location of task v_i
X	Execution location vector of all tasks in application G
σ^2	Noise power
Ψ	System bandwidth
Notation used in reinforcement learning	
a	Action
$\mathcal{A}$	Action space
$\mathcal{B}$	Experience buffer
$Q(s, a)$	Multi-objective Q-value vector of taking action a in state s
$\mathbf{r}(s, a)$	Vector reward function
R	Return
s	State
$\mathcal{S}$	State space
$\mathcal{W}$	Set of encountered preferences
γ	Discount factor
Ω	Preference space

3.1. Local computing

Let CT^l_i denote the completion time of task v_i if v_i is executed locally on the MD. Let CT^t_i be the completion time when the input data of v_i are completely transmitted to edge server x_i via the wireless uplink channel. Let CT^s_i represent the completion time of v_i on edge server x_i if v_i is offloaded to x_i. Let CT^r_i be the completion time when the MD completely receives the output data of v_i from edge server x_i via the wireless downlink channel. We set $CT^t_i = CT^s_i = CT^r_i = 0$, if v_i is executed locally on the MD; we set $CT^l_i = 0$ if v_i is offloaded to x_i.

Task v_i cannot be executed unless all its immediate predecessors are completed. If v_i is to be executed on the MD, the ready time of v_i, RT^l_i, is defined as

$$RT^l_i = \max\{\max\{CT^l_j, CT^r_j\} | v_j \in pre(v_i)\}, \tag{1}$$

where task $v_j \in pre(v_i)$ is an immediate predecessor of v_i. If v_j is executed on the MD, we set $CT^r_j = 0$; if v_j is executed on edge server x_i, we have $CT^l_j = 0$.

Note that task v_i cannot start its execution at the ready time, RT^l_i, if the MD is busy with executing other task at that time. This means that the actual start time of v_i is not earlier than its ready time. The delay for executing v_i on the MD, T^l_i, is calculated by c_i/f^l, where c_i is the number of CPU cycles required to execute v_i and f^l is the computing capability of the MD. The corresponding energy consumption is $E^l_i = \kappa \cdot c_i \cdot (f^l)^2$ [11], where κ is the effective capacitance coefficient dependent on the chip architecture used.

3.2. Edge computing

Assume that task v_i is to be offloaded to edge server $x_i \in \mathcal{M}$. The MD needs to transmit the input data of v_i to x_i through the wireless uplink channel. The ready time for transmitting the input data of the task, RT^t_i, is defined as

$$\begin{aligned} RT^t_i = \max\{&\{\max\{CT^l_j, CT^r_j\} | v_j \in pre(v_i), x_j \neq x_i\}, \\ &\max\{CT^t_j | v_j \in pre(v_i), x_j = x_i\}\}, \end{aligned} \tag{2}$$

where $CT^l_j = 0$ means v_j is offloaded to x_j for execution. If v_j is executed on the MD, we have $CT^t_j = 0$ and $CT^r_j = 0$.

According to the Shannon–Hartley theorem [26], the achievable uplink transmission rate from the MD to edge server x_i is

$$\eta_{x_i} = \Psi \cdot \log_2\left(1 + \frac{p^t \cdot g_{x_i}}{\sigma^2}\right), \tag{3}$$

where Ψ, p^t, and σ^2 are the system bandwidth, transmission power of the MD, and noise power, respectively. g_{x_i} is the channel gain between the MD and x_i. Besides, we suppose the wireless channels are symmetric, i.e., the achievable uplink and downlink transmission rates are the same. Note that the wireless uplink or downlink transmission rate may change when offloading different tasks in an application because of the time-varying wireless channel qualities. This will increase or decrease the transmission delay between an edge server and the MD, thus affecting the offloading delay and energy consumption of the MD. Therefore, when designing computing offloading policies, the time-varying wireless channel qualities should be considered. The delay for transmitting the input data of task v_i to x_i is $T^t_i = d_i/\eta_{x_i}$, where d_i is the input data size of v_i. The corresponding energy consumption for transmission, E^t_i, is calculated by

$$E^t_i = p^t \cdot T^t_i. \tag{4}$$

Once the input data of task v_i are received by edge server x_i via the wireless uplink channel, v_i needs to prepare for its execution on x_i. The ready time for executing v_i on x_i, RT^s_i, is obtained by

$$RT^s_i = \max\{CT^t_i, \max\{CT^s_j | v_j \in pre(v_i), x_j = x_i\}\}, \tag{5}$$

where v_j is an immediate predecessor of v_i. If v_j is executed on the MD, we have $CT^s_j = 0$. The delay for executing v_i on x_i is $T^s_i = c_i/f^s_{x_i}$, where $f^s_{x_i}$ is the computing capability of x_i.

Edge server x_i can start to transmit the output data of task v_i back to the MD, immediately after task v_i is completed. Thus, the ready time for edge server x_i to transmit the output data of v_i to the MD, is equal to the time when the task is completed on the edge server. The delay for the MD to receive the output data of v_i from x_i, is defined as $T^r_i = q_i/\eta_{x_i}$, where q_i is the output data size after executing v_i. The corresponding energy consumption for receiving the output data, E^r_i, is defined as

$$E^r_i = p^r \cdot T^r_i, \tag{6}$$

where p^r is the receiving power of the MD.

Similar to [10,15], we neglect the propagation delay of wireless uplink and downlink channels because of the short distance between the MD and edge servers. Thus, the network latency can be obtained by $T^t_i + T^s_i + T^r_i$ when the MD offloads task v_i to edge server x_i.

Based on Eqs. (4) and (6), the total energy consumption for offloading task v_i to edge server x_i is

$$E^o_i = E^t_i + E^r_i. \tag{7}$$

In addition, when task v_i is to be executed on edge server x_i, the MEC system operator will charge the MD because it rents the computing resources of the edge server to execute the task. The usage charge is calculated based on the price per time unit at the computing capability $f_{x_i}^s$, defined as $C(f_{x_i}^s) = e^{-\alpha} \cdot (e^{f_{x_i}^s} - 1) \cdot \beta$ [39], where α and β are two coefficients. The usage charge required to execute task v_i on edge server x_i is obtained by

$$C_i^s = C(f_{x_i}^s) \cdot T_i^s. \tag{8}$$

3.3. Problem formulation

The ODT problem formulated in this paper aims to simultaneously minimize the application completion time, energy consumption of the MD, and usage charge for edge computing in the MEC system.

Given application G, its application completion time is defined as

$$T(G) = \max\{\max\{CT_i^l, CT_i^r\} | v_i \in end(G)\}, \tag{9}$$

where $\max\{CT_i^l, CT_i^r\}$ stands for the completion time of end task v_i. It equals to CT_i^l, if v_i is executed on the MD; it equals to CT_i^r, otherwise.

The energy consumption of the MD is the summation of the energy consumption incurred by local and edge computing, defined as

$$E(G) = \sum_{i=1}^{N} E_i = \sum_{i=1}^{N} \left(E_i^l \cdot \mathbf{1}_{(x_i=0)} + E_i^o \cdot \mathbf{1}_{(x_i \in \mathcal{M})} \right), \tag{10}$$

where E_i is the energy consumption for executing task v_i. $\mathbf{1}_{(\Delta)}$ is an indicator function that equals 1 if condition Δ is satisfied; $\mathbf{1}_{(\Delta)}$ is equal to 0, otherwise.

The usage charge for edge computing is the summation of the usage charge for executing v_i on edge server x_i, defined as

$$C(G) = \sum_{i=1}^{N} C_i = \sum_{i=1}^{N} C_i^s \cdot \mathbf{1}_{(x_i \in \mathcal{M})}, \tag{11}$$

where C_i is the usage charge for executing task v_i.

We formulate the ODT problem concerned in this paper as a multi-objective optimization problem (MOP) with $T(G)$, $E(G)$, and $C(G)$ minimized simultaneously, subject to the dependency constraints between tasks, as follows.

$$\min_{x_i} \ (T(G), E(G), C(G)) \tag{12}$$

s.t.

$$\begin{aligned}
&C1: x_i \in \{0, 1, \ldots, M\}, \forall i \in \{1, \ldots, N\},\\
&C2: CT_j^l \le RT_i^l, \forall v_i \in V, v_j \in pre(v_i),\\
&C3: CT_j^r \le RT_i^l, \forall v_i \in V, v_j \in pre(v_i),\\
&C4: CT_i^t \le RT_i^s, \forall v_i \in V,\\
&C5: CT_j^s \le RT_i^s, \forall v_i \in V, v_j \in pre(v_i), x_j = x_i.
\end{aligned}$$

Constraint C1 denotes the task execution location constraint, specifying where task v_i is executed, i.e., the MD or which edge server. If v_i is to be executed on the MD, constraints C2 and C3 apply and otherwise, constraints C4 and C5 apply. C2 and C3 specify v_i cannot be launched until all its immediate predecessors are completed, i.e., those executed on the MD must be completed while those on the edge server(s) must completely return their output data to the MD. If v_i is to be executed on edge server x_i, C4 specifies that the input data of v_i must be ready on x_i before launching v_i; C5 specifies that v_i cannot be executed until all its immediate predecessors executed on x_i must be completed.

The new ODT problem is featured with multiple conflicting objectives. For example, although offloading can decrease the energy consumption of the MD, it incurs additional usage charge for edge computing. Besides, the preferences between objectives may change over time in dynamic MEC systems. For instance, an MD with lower battery level is more likely to give preference to energy consumption reduction, while it is more important to shorten the application completion time of delay-sensitive applications. Unfortunately, traditional SORLs cannot well handle the ever-changing preferences between objectives as they aggregate multiple objectives into one by scalarization techniques. This is why we attempt to adopt MORL-DWS, an emerging MORL algorithm, to address the multi-objective ODT problem concerned in this paper.

4. Overview of MOMDP and MORL-DWS

An MOMDP can be denoted by tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathbf{r}, \Omega, f \rangle$ [20], where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}$ is the state transition probabilities matrix, $\mathbf{r}(s, a)$ is the vector reward function, Ω is the preference space, and f is the scalarization function mapping the multi-objective value V^π of a policy π to a scalar value, i.e., the user utility. In this paper, we focus on a linear function f, where $f(V^\pi, \mathbf{w}) = \mathbf{w} \cdot V^\pi, \mathbf{w} \in \Omega$. We observe that if $\mathbf{w}$ is fixed, the MOMDP collapses into a standard MDP.

In the MOMDP, a policy is a specific state-to-action mapping, $\pi : \mathcal{S} \to \mathcal{A}$. An agent learns an optimal policy that maximizes the expected return by interacting with the corresponding environment. The action-value function for policy π, denoted by $\mathbf{Q}_\pi(s, a)$, is the value of taking action a in state s. This function is defined in Eq. (13) as the expected return starting from s, taking a, and thereafter following policy π.

$$\begin{aligned}
\mathbf{Q}_\pi(s, a) &= \mathbb{E}_\pi \left[\mathbf{R}_t | s_t = s, a_t = a \right] \\
&= \mathbb{E}_\pi \left[\sum_{k=t}^{N} \gamma^{k-t} \cdot \mathbf{r}_k \middle| s_t = s, a_t = a \right],
\end{aligned} \tag{13}$$

where $\mathbf{R}_t$ is the cumulative discounted reward at time step t, discount factor $\gamma \in [0, 1]$ represents the present value of future rewards, $\mathbf{r}_t$ is the immediate reward received at time step t. Because each component of $\mathbf{r}_t$ corresponds to one objective, $\mathbf{Q}_\pi(s, a)$ is a multi-objective Q-value vector.

MORLs have attracted increasingly more research attention [17]. Among them, MORL-DWS is one of the most popular MORLs and has been applied to various MOPs because it can generalize dynamic preferences by learning weight-dependent multi-objective Q-value vectors [19]. In MORL-DWS, a Q-network (also called conditioned network) is used to tackle high-dimensional inputs problem. This network takes both the current environment state and a preference $\mathbf{w} \in \Omega$ as input, and it outputs multi-objective Q-value vectors. Another advantage of MORL-DWS is that it prevents the network from overfitting to the current region of the preference space and forgetting previously learned policies. This is because the sampled transitions are trained on both the current preference and a random preference previously encountered. Note that the encountered preference is selected from the set of preferences the agent has experienced since it started learning. In each time step, a mini-batch of transitions is used to update the Q-network. For an arbitrary transition $(s_j, a_j, \mathbf{r}_j, s_{j+1})$, its loss is calculated by Eq. (14).

$$L_j = \frac{1}{2} \left[|\mathbf{y}_j - \mathbf{Q}(s_j, a_j; \mathbf{w}_k)| + |\mathbf{y}_j' - \mathbf{Q}(s_j, a_j; \mathbf{w}_j)| \right], \tag{14}$$

$$\mathbf{y}_j = \mathbf{r}_j + \gamma \bar{\mathbf{Q}} \left(s_{j+1}, \underset{a \in \mathcal{A}}{\operatorname{argmax}} \mathbf{Q}(s_{j+1}, a; \mathbf{w}_k) \mathbf{w}_k; \mathbf{w}_k \right), \tag{15}$$

$$\mathbf{y}_j' = \mathbf{r}_j + \gamma \bar{\mathbf{Q}} \left(s_{j+1}, \underset{a \in \mathcal{A}}{\operatorname{argmax}} \mathbf{Q}(s_{j+1}, a; \mathbf{w}_j) \mathbf{w}_j; \mathbf{w}_j \right), \tag{16}$$

Table 5
Task execution delay (Sec.)

Task	Mobile device	Edge server 1	Edge server 2
v_1	5	2	1
v_2	6	1	1
v_3	8	3	2
v_4	7	2	1
v_5	5	2	1
v_6	8	4	2
v_7	6	3	2
v_8	6	1	1
v_9	7	2	2

where $\mathbf{w}_k$ and $\mathbf{w}_j$ are the current and encountered preferences, respectively, and $\mathbf{Q}(s_j, a_j; \mathbf{w}_k)$ is the network's Q-value vector for action a_j in state s_j and current preference $\mathbf{w}_k$.

5. The proposed MORL for the ODT problem

This section proposes an improved MORL-DWS with TS scheme to solve the ODT problem, namely, MORL-ODT. We first introduce the task execution process by an example, including the local and edge computing. Then, we model the MOMDP for the ODT problem. Finally, the procedure of the proposed MORL-ODT is given in detail.

5.1. Example of task execution process

To meet the dependency constraints between tasks in application G, we sort tasks in descending order according to their rank values, defined as

$$rank(v_i) = \begin{cases} T_i^l, & \text{if } v_i \in end(G) \\ T_i^l + \max_{v_j \in suc(v_i)}\{rank(v_j)\}, & \text{if } v_i \notin end(G) \end{cases} \quad (17)$$

where T_i^l denotes the delay for executing task v_i on the MD and $suc(v_i)$ is the set of immediate successors of v_i. We regard the sorted order as the execution order vector, represented by $O = (u_1, \ldots, u_N)$, where u_t stands for the tth task to be executed in O. All tasks in G are executed following their order in O.

Taking the DAG in Fig. 2 as an example, we briefly explain the task execution process. Application G consists of nine tasks, $v_1, \ldots, v_9$. Assume there are two edge servers in the MEC system, i.e., we have $M = 2$. For simplicity, we set the delay for transmitting the input data of task v_i to edge server x_i, $T_i^t = 1$ s, and the delay for receiving the output data of task v_i from edge server x_i, $T_i^r = 1$ s. We set both the transmission and receiving powers to 0.1 Watt and the price per time unit for each edge server to 0.01 \$, respectively. We set the power consumption of the MD to 1 Watt.

Table 5 shows the execution delays of tasks on the MD and two edge servers. According to Eq. (17), we obtain the execution order vector $O = (v_1, v_3, v_2, v_4, v_5, v_6, v_7, v_8, v_9)$. Let us take v_1 and v_2 as an example. They are in the 1st and 3rd positions for execution in O, respectively. So, we have $u_1 = v_1$ and $u_3 = v_2$. Assume we have $X = (1, 1, 2, 0, 1, 2, 2, 0, 1)$ as the execution location vector, where x_i $(i = 1, \ldots, 9)$ is the execution location of v_i.

Fig. 3 shows the resultant task execution process based on the given O, X, and Table 5. For example, task v_6 is offloaded to edge server 2 for execution. Its ready time and completion time are $RT_6^t = 9$ s and $CT_6^r = 13$ s, respectively. According to Fig. 3, we have the application completion time, $T(G) = CT_9^r = 21$ s, the energy consumption of the MD, $E(G) = 14.4$ J, and the usage charge for edge computing, $C(G) = 0.13$ \$.

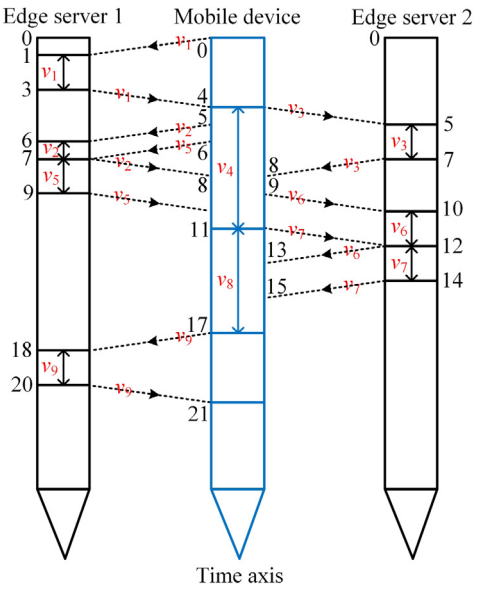

Fig. 3. Task execution process.

5.2. Modeling of MOMDP

(1) State space:

$$\mathcal{S} = \{s_t | s_t = (t-1, u_t, f^l, \mathbf{f}_t, \mathbf{g}_t), t = 1, \ldots, N\}, \quad (18)$$

where s_t is the state of the MEC system environment at time step t. In state s_t, $t-1$ denotes the number of tasks that have been executed so far; u_t is the tth task to be executed in the execution order vector O; f^l is the computing capability of the MD; $\mathbf{f}_t = (f_{1,t}^s, \ldots, f_{M,t}^s)$ represents the computing capabilities of the M edge servers at time step t; $\mathbf{g}_t = (g_{1,t}, \ldots, g_{M,t})$ stands for the channel gains between the MD and the M edge servers at time step t.

(2) Action space:

$$\mathcal{A} = \{a_t | a_t \in \{0, 1, \ldots, M\}, t = 1, \ldots, N\}, \quad (19)$$

where a_t is an action the MD takes in state s_t. In other words, a_t is the execution location (i.e., x_t) of task $u_t \in O$. If a_t equals to 0, u_t is executed on the MD, and otherwise, u_t is offloaded to edge server a_t for execution.

(3) Reward function: The ODT problem aims to simultaneously minimize the application completion time, $T(G)$, energy consumption of the MD, $E(G)$, and usage charge for edge computing, $C(G)$. Hence, after taking action a_t in state s_t, we define the received vector-valued reward as

$$\mathbf{r}_t = (r_t^T, r_t^E, r_t^C), \quad (20)$$

where r_t^T, r_t^E, and r_t^C are the scalar rewards of objectives $T(G)$, $E(G)$, and $C(G)$, respectively.

The following describes the three scalar rewards r_t^T, r_t^E, and r_t^C. Let G_t denote a subgraph consisting of the first t tasks in the execution order vector O. We calculate the completion time of G_t, $T(G_t)$, for example, $T(G_4) = 11$ s in Fig. 3. To minimize $T(G)$, we set r_t^T to $(T(G_{t-1}) - T(G_t))$, denoting the negative increment of the completion time of G_{t-1} after executing $u_t \in O$ based on action a_t. $T(G_0) = 0$ means no task is executed. As to reward r_t^E, we first calculate the energy consumption, E_t, incurred by executing u_t. Then, we set r_t^E to the negative value of E_t, i.e., $r_t^E = -E_t$. Similarly, we set $r_t^C = -C_t$, where C_t is the usage charge for executing u_t.

Based on the reward $\mathbf{r}_t$, we can obtain the return, which is the summation of the discounted reward generated at each time step over the long run. Let $\mathbf{R}_t = (R_t^T, R_t^E, R_t^C)$ denote the return of r_t^T, r_t^E, and r_t^C at time step t, defined as

$$R_t^T = \sum_{k=t}^{N} \gamma^{k-t} r_k^T = \sum_{k=t}^{N} \gamma^{k-t} (T(G_{k-1}) - T(G_k)), \tag{21}$$

$$R_t^E = \sum_{k=t}^{N} \gamma^{k-t} r_k^E = -\sum_{k=t}^{N} \gamma^{k-t} E_k, \tag{22}$$

$$R_t^C = \sum_{k=t}^{N} \gamma^{k-t} r_k^C = -\sum_{k=t}^{N} \gamma^{k-t} C_k. \tag{23}$$

We calculate the return $\mathbf{R}_1$, with the discount factor γ and the current time step t both set to 1. Note that $t = 1$ means all tasks in application G will be executed following their order in O. According to Eqs. (21)–(23), we have $R_1^T = -T(G_N) = -T(G)$, $R_1^E = -E(G)$, and $R_1^C = -C(G)$, i.e., $\mathbf{R}_1 = (R_1^T, R_1^E, R_1^C) = (-T(G), -E(G), -C(G))$. So, to maximize the expected return $\mathbb{E}[\mathbf{R}_1]$ is equivalent to minimizing $T(G)$, $E(G)$, and $C(G)$, simultaneously.

5.3. The MORL-ODT algorithm

The framework of MORL-ODT is shown in Fig. 4. It is composed of two interacting components: the observed MEC system environment and the learning agent. In the MEC system environment, there are M edge servers and one MD, as shown in Fig. 1. The learning agent is used to interact with the environment, taking an action for each task to be executed. The agent involves in two phases, i.e., execution and training. As shown in Fig. 4, the Q-network in the execution phase is exactly the same as the one in the training phase. In the execution phase, the agent observes the states of the MEC system by consistently interacting with the environment and takes actions to schedule tasks of a given application. In the training phase, at each time step, a mini-batch of transitions is randomly sampled from the experience buffer to train the Q-network.

The sequence diagram of the MORL-ODT framework is depicted in Fig. 5. It shows the interaction among different components of the proposed framework. First, state s observed by the agent and preference $\mathbf{w}_k$ sampled from preference space Ω are fed into the Q-network, which outputs action a. The agent takes the action to schedule a task in a given application, and it receives the next state s' and reward $\mathbf{r}$ from the MEC system environment. The transition $(s, a, \mathbf{r}, s')$ is stored in the experience buffer. Then, a mini-batch of transitions is randomly sampled from the experience buffer. An encountered preference is selected using the TS scheme from the encountered preference set. Next, both the mini-batch and encountered preference are sent to the data preparation (DP) component. The DP component concatenates states and preferences, such as $(s, \mathbf{w}_k)$ and $(s', \mathbf{w}_k)$. The concatenated results are input to the Q-network and target Q-network, respectively. Based on the outputs of the two networks, the loss calculation component calculates the loss for updating the parameters of the Q-network. Finally, the target Q-network is synchronized using the parameters of the Q-network.

The pseudo-code of MORL-ODT is shown in Algorithm 1. This algorithm is based on MORL-DWS, with a tournament selection (TS) scheme incorporated. After initializing the Q-network, the agent starts interacting with the MEC system environment. At each episode, the current preference $\mathbf{w}_k$ is randomly sampled from the preference space Ω. If $\mathbf{w}_k$ is not in the set of encountered preferences, $\mathcal{W}$, we add it together with its corresponding episode value k to $\mathcal{W}$. If $\mathbf{w}_k$ is in $\mathcal{W}$, the episode value of $\mathbf{w}_k$ is updated as k. That we add and update the episode values for

Algorithm 1 MORL-ODT for ODT problem

1: Initialize Q-network $\mathbf{Q}(s, a; \mathbf{w})$ with random parameters θ;
2: Initialize target Q-network $\bar{\mathbf{Q}}(s, a; \mathbf{w})$ with parameters $\bar{\theta} = \theta$;
3: Initialize preference space Ω;
4: Set $\mathcal{B} = \emptyset$ and $\mathcal{W} = \emptyset$;
5: **for** $k = 1, ..., EPS_{max}$ **do** // for each episode
6: Sample a preference $\mathbf{w}_k$ from Ω;
7: **if** $\mathbf{w}_k$ is not in $\mathcal{W}$ **then**
8: Add $\mathbf{w}_k$ with episode value k to $\mathcal{W}$;
9: **else**
10: Update the episode value of $\mathbf{w}_k$ as k;
11: **end if**
12: Observe state s_1;
13: **for** time step $t = 1, ..., N$ **do** // for each task in O
14: Select action a_t using Eq. (24); // ε-greedy
15: Take action a_t and observe $\mathbf{r}_t$ and s_{t+1};
16: Store transition $(s_t, a_t, \mathbf{r}_t, s_{t+1})$ in $\mathcal{B}$;
17: Sample a mini-batch of transitions from $\mathcal{B}$;
18: **for** each sampled transition $(s_j, a_j, \mathbf{r}_j, s_{j+1})$ **do**
19: Select an encountered preference $\mathbf{w}_j$ from $\mathcal{W}$ using TS scheme;
20: **if** transition is terminal **then**
21: Set $\mathbf{y}_j = \mathbf{y}_j' = \mathbf{r}_j$;
22: **else**
23: Calculate $\mathbf{y}_j$ and $\mathbf{y}_j'$ using Eqs. (15) and (16), respectively;
24: **end if**
25: **end for**
26: Perform gradient descent step on Eq. (14) to update Q-network;
27: Every $\bar{N}$ time steps, synchronize target Q-network, i.e., $\bar{\theta} = \theta$;
28: Anneal ε;
29: **end for**
30: **end for**

preferences in $\mathcal{W}$ is because it is convenient for us to select proper encountered preferences using their episode values. In this way, the Q-network takes care of both the current and encountered preferences during training.

The agent selects an action according to the ε-greedy exploration, defined as

$$a_t = \begin{cases} \text{a randomly selected action from } \mathcal{A}, & \text{with prob. } \varepsilon \\ \text{argmax}_{a \in \mathcal{A}} \mathbf{Q}(s_t, a_t; \mathbf{w}_k)\mathbf{w}_k, & \text{with prob. } 1 - \varepsilon \end{cases} \tag{24}$$

where 'prob.' is the short-form for probability, and $\varepsilon \in [0, 1]$ is the probability of random exploration and annealed over time steps. After taking action a_t, the MEC system environment responds with a transition to the next state s_{t+1}, and provides the agent with reward $\mathbf{r}_t$. The transition $(s_t, a_t, \mathbf{r}_t, s_{t+1})$ is added to the experience buffer, $\mathcal{B}$.

To train the Q-network, we randomly sample a mini-batch of transitions from $\mathcal{B}$. For each transition in the mini-batch, we select an encountered preference from $\mathcal{W}$ to maintain the learned policies. The original MORL-DWS randomly selects an encountered preference from $\mathcal{W}$. However, this method selects preferences at the same frequency regardless of their importance, resulting in that the learned policies are forgotten easily. This is why we propose the TS scheme (step 19 in Algorithm 1) to select those more significant preferences at higher probabilities. In this scheme, for each preference, its episode value is used to reflect its significance and this value is updated over episodes (steps 7–11 in Algorithm 1). The smaller the values, the more important the preferences.

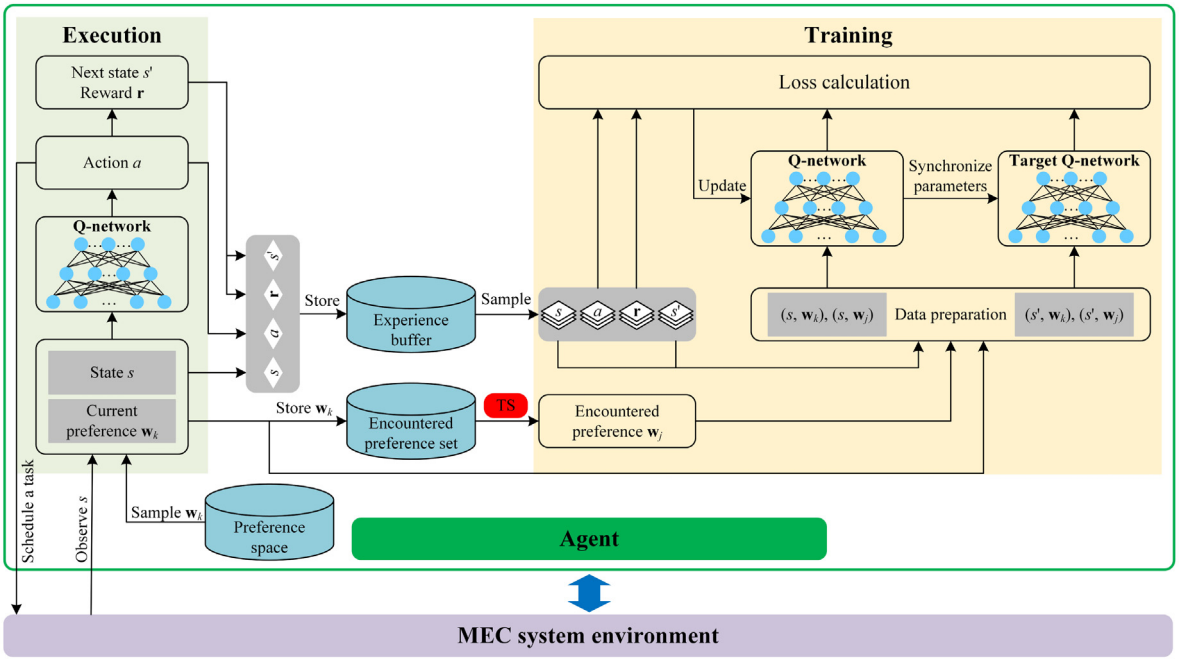

Fig. 4. Framework of MORL-ODT.

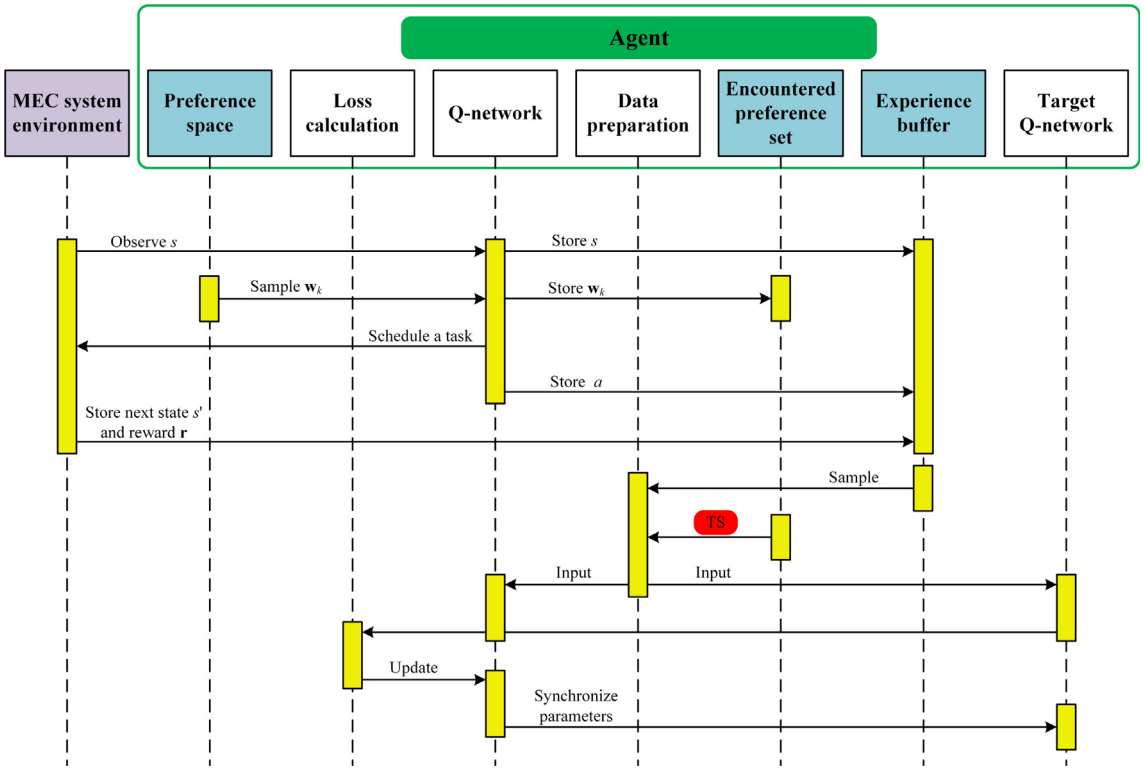

Fig. 5. Sequence diagram of the MORL-ODT framework.

A preference with a smaller episode value indicates that the Q-network may not get trained on the preference within the past time steps, which is difficult for the network to memorize the corresponding policy. The Q-network is more likely to maintain the learned policies if preferences with smaller episode values are taken as input more frequently.

The following briefly introduces how the TS scheme works. Once invoked, this scheme randomly selects H encountered preferences from $\mathcal{B}$, where H is the tournament size. Then, their episode values are compared and the preference with the smallest episode value is returned.

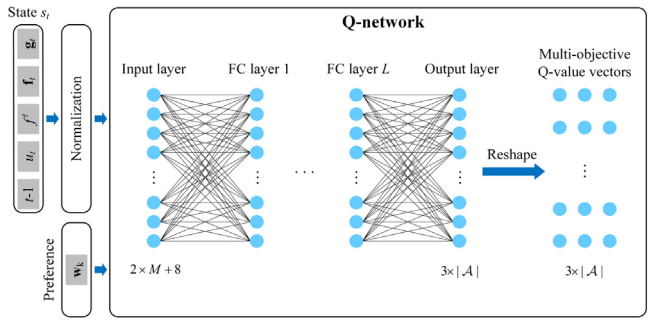

Fig. 6. Network architecture.

5.4. Network architecture

Fig. 6 illustrates the Q-network architecture used in our implementation. The Q-network consists of an input, an output and L fully connected (FC) layers, where L is the number of FC layers. The input layer has $2 \times M + 8$ neurons, where M is the number of edge servers. The output layer has $3 \times |\mathcal{A}|$ neurons and its output is reshaped into an $3 \times |\mathcal{A}|$ matrix. Each row in the matrix corresponds to a multi-objective Q-value vector of action $a \in \mathcal{A}$. The matrix multiplies on the right by a preference to apply traditional action selection schemes, such as, the ε-greedy exploration.

State $s_t = (t-1, u_t, f^l, \mathbf{f}_t, \mathbf{g}_t)$ goes through normalization before it is fed into the network as its elements have different value ranges. For example, the number of tasks that have been executed so far, $t-1$, is an integer between 0 to N. While the computing capability of edge server m, $f^s_{m,t} \in \mathbf{f}_t$, is in the range of [2, 6] GHz. The widely used Min–Max normalization is adopted in the preprocessing process. The preprocessed state and the preference are concatenated together and fed into the input layer of the Q-network.

5.5. Complexity analysis

We first analyze the time complexity of MORL-ODT shown in Algorithm 1 in the training process. The time complexity mainly depends on the Q-network training complexity of MORL-ODT. According to Fig. 6, the Q-network consists of an input, an output, and L fully connected (FC) layers. The input layer has $2 \times M + 8$ neurons, where M is the number of edge servers. Let n_i denote the number of neurons of the ith FC layer. The output layer has $3\times|\mathcal{A}|$ neurons, where $|\mathcal{A}| = M+1$. Note that we have $n_0 = 2\times M+8$ and $n_{L+1} = 3\times|\mathcal{A}|$. Let EPS_{max} and N denote the maximum number of episodes and the number of time steps per episode, respectively. Let $|\bar{\mathcal{B}}|$ denote the batch size. The time complexity of MORL-ODT for training is $O(EPS_{max} \times N \times (\sum_{i=1}^{L+1} n_{i-1} \times n_i) \times |\bar{\mathcal{B}}|)$.

Once the training process is finished, the trained Q-network is used to make offloading decisions via network inference. In each time step, the execution location of each task in G can be generated through the Q-network. Therefore, based on the task number of G and the inference time of the Q-network, the time complexity of MORL-ODT for testing is $O(N \times \sum_{i=1}^{L+1} n_{i-1} \times n_i)$.

6. Simulation results and discussion

We develop a python simulator running on Google TensorFlow 2.2 for performance evaluation. In the experiment, we assume there are $M = 3$ edge servers to provide edge computing services to the MD. The computing capability of each edge server is randomly generated in the range of [2, 6] GHz. The channel gain states between the MD and the edge servers are from a common finite set $\{-18, -16, -14, -12, -10, -8\}$ (dB). The transitions of these states occur across the time steps [40]. For an arbitrary channel, its next gain state is randomly selected from the finite set above. There are two FC layers in the Q-network, i.e., $L = 2$. Each layer has 64 neurons, with the rectified linear units (ReLU) as activation function and Adam as the optimizer. The discount factor γ is set to 0.99. We set the maximum number of episodes, $EPS_{max} = 2000$. The parameter of the ε-greedy exploration, ε, is annealed from 1 to 0.01 over time steps, with a decay rate of 0.995. Other parameter values used in the experiment are listed in Table 6. The results are obtained by running each algorithm 10 times, from which the statistics are collected and analyzed.

Table 6
Experimental Configuration.

Parameter	Value
Number of edge servers (M)	3
System bandwidth (Ψ)	0.6 MHz
Noise power (σ^2)	10^{-6} Watt
Computing capability of each edge server (f^s_m)	[2, 6] GHz
Computing capability of the MD (f^l)	0.6 GHz
Effective capacitance coefficient (κ)	10^{-26}
Transmission power of the MD (p^t)	0.5 Watt
Receiving power of the MD (p^r)	0.1 Watt

Table 7
Test Instances and Their Parameters.

Instance	N	fat	$density$	c_i (GHz)	d_i (Kb)	q_i (Kb)
Rnd-1	10	0.4	0.7	[0.1, 0.5]	[5000, 6000]	[500, 1000]
Rnd-2	10	0.6				
Rnd-3	20	0.4				
Rnd-4	20	0.6				
Rnd-5	30	0.4				
Rnd-6	30	0.6				

We use DAG to mimic an application with task dependency constraints. We adopt a well-known synthetic DAG generation program [41] to generate test applications with various task topologies and profiles. The parameters of a DAG are listed below.

- N: number of tasks in the DAG.
- fat: controls the width and height of the DAG.
- $density$: decides the number of edges between two levels of the DAG, i.e., the dependency constraints between tasks.
- c_i: number of CPU cycles required to execute task v_i in the DAG, $i = 1, \ldots, N$.
- d_i: input data size of task v_i in the DAG, $i = 1, \ldots, N$.
- q_i: output data size of task v_i in the DAG, $i = 1, \ldots, N$.

In our experiment, we use the following parameter values for random DAG generation, where six test instances are generated, as shown in Table 7.

- $N \in \{10, 20, 30\}$.
- $fat \in \{0.4, 0.6\}$. $fat = 0.4$ will induce a thin DAG with low task parallelism, whereas $fat = 0.6$ results in a fat DAG with a high degree of parallelism.
- $density = 0.7$. It generates large dependency constraints between tasks.
- c_i is randomly generated in the range of [0.1, 0.5] GHz.
- d_i is randomly generated in the range of [5000, 6000] Kb.
- q_i is randomly generated in the range of [500, 1000] Kb.

A weight vector (i.e., preference) generation method [42] is adopted to generate the preference space $\Omega = \{\mathbf{w}_1, \ldots, \mathbf{w}_{|\Omega|}\}$. For an arbitrary weight vector $\mathbf{w} \in \Omega$, each of its individual weight takes a value from

$$\left\{\frac{0}{\Gamma}, \frac{1}{\Gamma}, \ldots, \frac{\Gamma}{\Gamma}\right\},$$

where Γ is an integer. Let σ be the number of objectives. Parameters Γ and σ together determine the number of preferences in Ω, i.e., $C^{\sigma-1}_{\Gamma+\sigma-1}$. The weight vector generation method generates uniformly distributed weight vectors, mimicking various preferences between objectives. As the ODT problem has three objectives, we have $\sigma = 3$. We set $\Gamma = 16$ in the experiment. So, we have $|\Omega| = C^{3-1}_{16+3-1} = 153$. A promising MORL algorithm should generalize across the entire preference space, being well adapted to the dynamics of preferences.

6.1. Performance measures

(1) Regret: Similar to [19], we evaluate the performance of algorithms based on their regret. The regret is the difference between the optimal value and actual return, defined as

$$\begin{aligned}\Delta(\mathbf{w}, \mathbf{R}) &= \mathbf{w} \cdot \mathbf{V}^*_{\mathbf{w}} - \mathbf{w} \cdot \mathbf{R} \\ &= \mathbf{w} \cdot \mathbf{V}^*_{\mathbf{w}} - \mathbf{w} \cdot \sum_{k=1}^{N} \gamma^{k-1} \cdot \mathbf{r}_k,\end{aligned} \tag{25}$$

where $\mathbf{V}^*_{\mathbf{w}}$ is the optimal value for preference $\mathbf{w} \in \Omega$, $\mathbf{R}$ is the actual return, and $\{\mathbf{r}_1, \ldots, \mathbf{r}_N\}$ is the set of vector-valued rewards collected during an episode of length N. After each episode ends, we obtain the regret for the current preference $\mathbf{w}$ according to Eq. (25). Unlike the actual return, the regret allows for a common optimal value for $\mathbf{w}$, i.e., an optimal policy always has 0 regret.

(2) Adaptation error: To measure an MORL's adaptation to dynamic preferences, we compare the actual return $\mathbf{R}$ with the optimal value $\mathbf{V}^*_{\mathbf{w}}$, when the MORL is provided with a specific preference $\mathbf{w} \in \Omega$ during the test phase. The adaptation error (AE) is defined as the average relative error between $\mathbf{w} \cdot \mathbf{R}$ and $\mathbf{w} \cdot \mathbf{V}^*_{\mathbf{w}}$ for each $\mathbf{w}$ in Ω:

$$AE = \frac{1}{|\Omega|} \sum_{\mathbf{w} \in \Omega} \left| \frac{\mathbf{w} \cdot \mathbf{R} - \mathbf{w} \cdot \mathbf{V}^*_{\mathbf{w}}}{\mathbf{w} \cdot \mathbf{V}^*_{\mathbf{w}}} \right|, \tag{26}$$

where $|\Omega|$ is cardinality of Ω. A smaller AE indicates the corresponding MORL is more adaptive to scenarios with dynamic preferences.

(3) Comprehensive objective indicator: As the ODT problem is of three-objective, we design a comprehensive cost that reflects an MORL's overall performance, with the application completion time, energy consumption, and usage charge considered simultaneously. For each objective vector, we aggregate its three objective values into a comprehensive objective indicator via linear scalarization.

Given a preference $\mathbf{w} = (w_1, w_2, w_3)$, an MORL algorithm may obtain multiple objective vectors after a single run. Let $TEC^i_{\mathbf{w}}(G) = (T^i_{\mathbf{w}}(G), E^i_{\mathbf{w}}(G), C^i_{\mathbf{w}}(G))$ denote the ith objective vector for $\mathbf{w}$, $i = 1, \ldots, Z_{\mathbf{w}}$, where $Z_{\mathbf{w}}$ is the number of objective vectors obtained. We define the comprehensive objective indicator of $TEC^i_{\mathbf{w}}(G)$, $COI^i_{\mathbf{w}}$, as

$$\begin{aligned}COI^i_{\mathbf{w}} &= \mathbf{w} \cdot TEC^i_{\mathbf{w}}(G) \\ &= w_1 \cdot T^i_{\mathbf{w}}(G) + w_2 \cdot E^i_{\mathbf{w}}(G) + w_3 \cdot C^i_{\mathbf{w}}(G).\end{aligned} \tag{27}$$

We define the best objective vector for $\mathbf{w}$, $TEC^b_{\mathbf{w}}(G)$, as

$$TEC^b_{\mathbf{w}}(G) = (T^b_{\mathbf{w}}(G), E^b_{\mathbf{w}}(G), C^b_{\mathbf{w}}(G)), b = \underset{i \in \{1,\ldots,Z_{\mathbf{w}}\}}{\operatorname{argmin}} COI^i_{\mathbf{w}}, \tag{28}$$

where $T^b_{\mathbf{w}}(G)$, $E^b_{\mathbf{w}}(G)$, and $C^b_{\mathbf{w}}(G)$ are the best $T(G)$, $E(G)$, and $C(G)$ for $\mathbf{w}$, respectively. According to Eqs. (27) and (28), we obtain the best objective vector for each preference in Ω. After that, we calculate the average application completion time (AACT), average energy consumption (AEC), average usage charge (AUC), and average comprehensive objective indicator (ACOI), respectively, defined as

$$AACT = \frac{1}{|\Omega|} \sum_{\mathbf{w} \in \Omega} T^b_{\mathbf{w}}(G), \tag{29}$$

$$AEC = \frac{1}{|\Omega|} \sum_{\mathbf{w} \in \Omega} E^b_{\mathbf{w}}(G), \tag{30}$$

$$AUC = \frac{1}{|\Omega|} \sum_{\mathbf{w} \in \Omega} C^b_{\mathbf{w}}(G), \tag{31}$$

$$ACOI = \frac{1}{|\Omega|} \sum_{\mathbf{w} \in \Omega} COI^b_{\mathbf{w}}(G). \tag{32}$$

(4) Network usage: The usage of network resources depends on the size of transmitting the input data to edge servers and receiving the output data from edge servers within the application completion time. The NU is defined in Eq. (33).

$$NU = \frac{1}{|\Omega|} \sum_{\mathbf{w} \in \Omega} \frac{1}{T^b_{\mathbf{w}}(G)} \sum_{i=1}^{N} (T^t_i \cdot d_i + T^r_i \cdot q_i) \cdot \mathbf{1}_{(x_i \in \mathcal{M})}, \tag{33}$$

where $T^b_{\mathbf{w}}(G)$ is the best $T(G)$ for $\mathbf{w}$ in Ω. $\mathcal{M}$ is the set of edge servers. T^t_i denotes the delay for transmitting the input data of task v_i to edge server $x_i \in \mathcal{M}$, and T^r_i denotes the delay for the MD to receive the output data of v_i from x_i. d_i and q_i are the input and output data sizes, respectively. $\mathbf{1}_{(\Delta)}$ is an indicator function that equals 1 if condition Δ is satisfied; $\mathbf{1}_{(\Delta)}$ is equal to 0, otherwise. Note that if v_i is executed on the MD, no network resources are used when executing v_i.

6.2. Parameter study of MORL-ODT

We take Rnd-1 in Table 7 as an example to study the impact of parameters on the performance of MORL-ODT, including the learning rate, batch size, buffer size, and tournament size. The cumulative regret of MORL-ODT in Rnd-1 is illustrated in Fig. 7. Fig. 7(a) shows the impact of the learning rate in the Adam optimizer on the algorithm performance. It is obvious that either a too small or a too large learning rate leads to a deteriorated cumulative regret curve. Thus, we hereafter fix the learning rate to 0.001. As shown in Fig. 7(b), a small batch size cannot efficiently cover the majority of transitions stored in the experience buffer, while a large batch size degrades the cumulative regret quality as the old transitions in the buffer are frequently selected and used. So, we hereafter set the batch size to 32. In Fig. 7(c), the smallest buffer size causes the worst performance because it limits the number of experiences stored. On the other hand, a large buffer size causes a poor performance due to that many old transitions cannot be updated timely. Hence, we hereafter set the buffer size to 3000. As shown in Fig. 7(d), the performance of MORL-ODT is not sensitive to the tournament size. To reduce the operational complexity of the TS scheme, we hereafter set the tournament size to 2.

6.3. Overall performance evaluation

To thoroughly study the performance of the proposed MORL-ODT, we compare it with three state-of-the-art MORLs in terms of regret and adaptation error in six test instances, as listed below.

- Naive [17]: This method designs an overall user utility by aggregating multiple single-objective Q-values into a synthetic function. The actions are selected using the synthetic function.

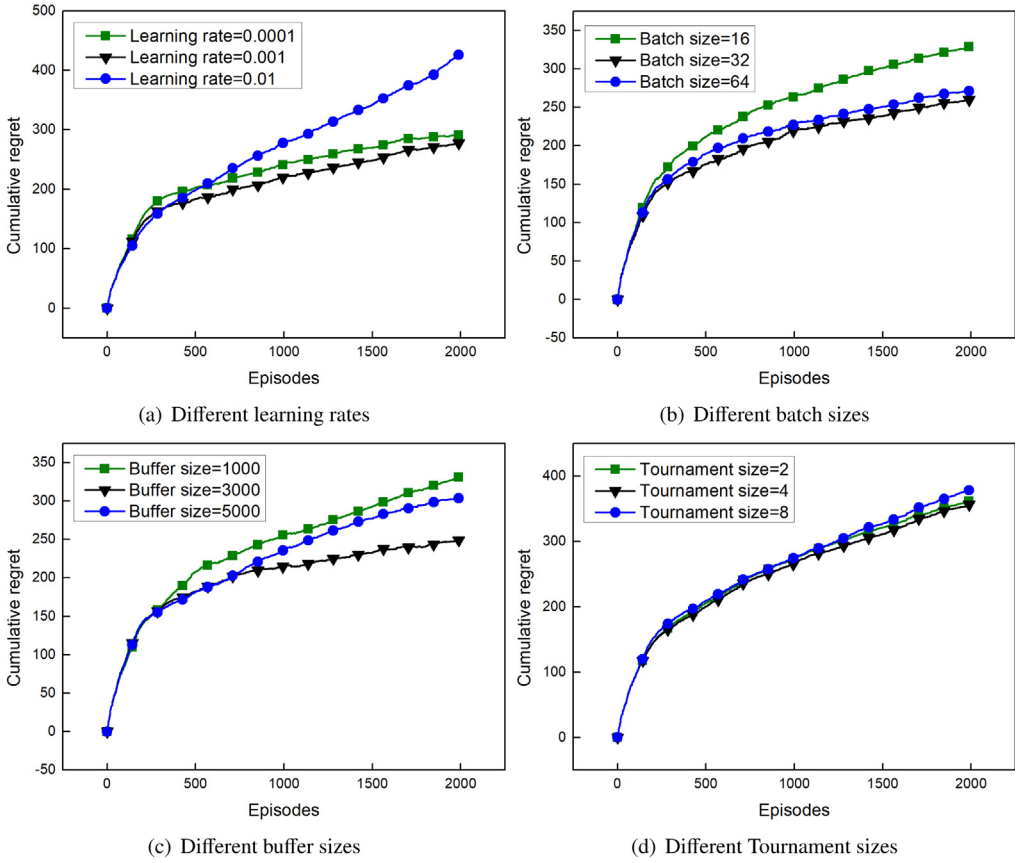

Fig. 7. Cumulative regret of MORL-ODT under different parameters in Rnd-1.

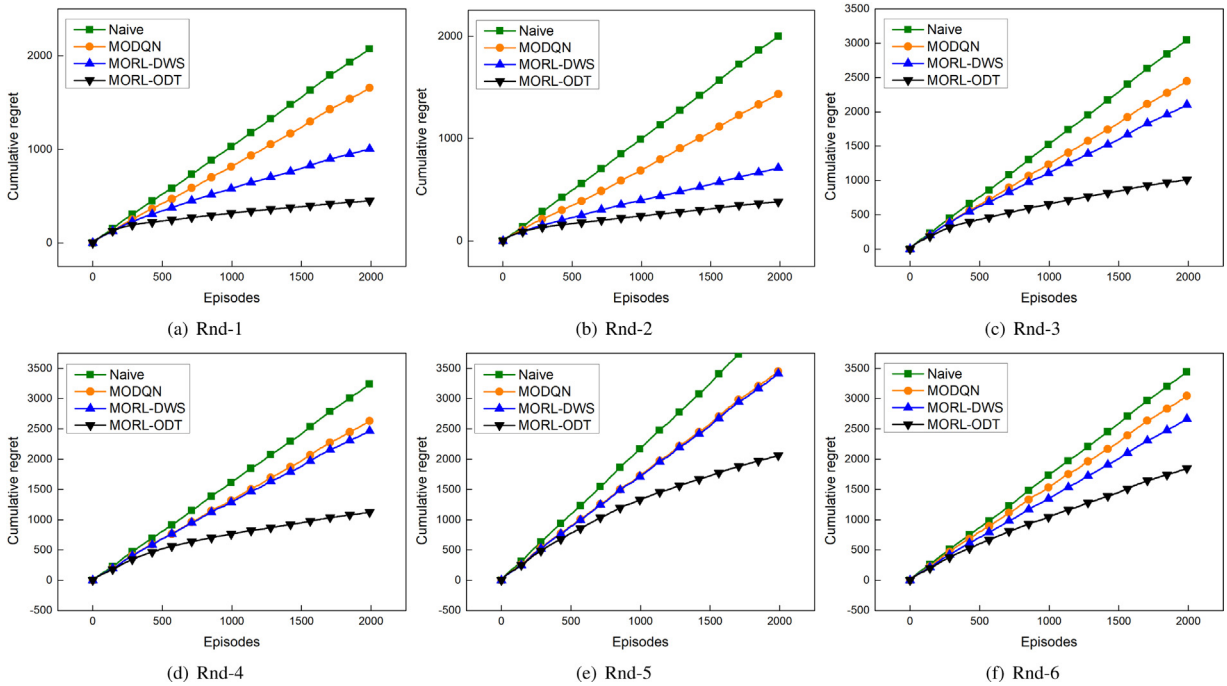

Fig. 8. Cumulative regret of four algorithms.

- MODQN [43]: Multi-objective deep Q-network method proposed to learn the vector-valued Q-function by scalarized deep Q-learning. The Q-network is only trained on the current preference.
- MORL-DWS [19]: MORL with dynamic weight setting developed to address dynamic preference problems. MORL-DWS generalizes across preference changes by learning multi-objective Q-value vectors.

Table 8
Results of Student's t-test based on AER and AE.

	Algorithm-A ↔ Algorithm-B	Rnd-1	Rnd-2	Rnd-3	Rnd-4	Rnd-5	Rnd-6
AER	MORL-ODT ↔ Naive	+	+	+	+	+	+
	MORL-ODT ↔ MODQN	+	+	+	+	+	+
	MORL-ODT ↔ MORL-DWS	+	+	+	+	+	+
	Algorithm-A ↔ Algorithm-B	Rnd-1	Rnd-2	Rnd-3	Rnd-4	Rnd-5	Rnd-6
AE	MORL-ODT ↔ Naive	+	+	+	+	+	+
	MORL-ODT ↔ MODQN	+	+	+	+	+	+
	MORL-ODT ↔ MORL-DWS	+	+	+	+	+	+

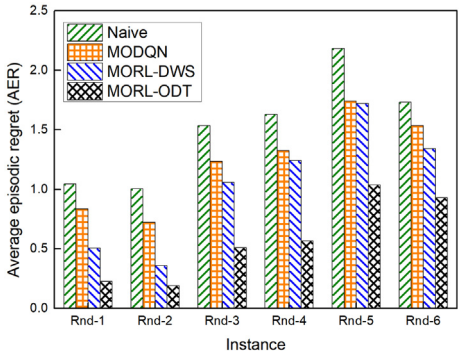

Fig. 9. Average episodic regret of four algorithms.

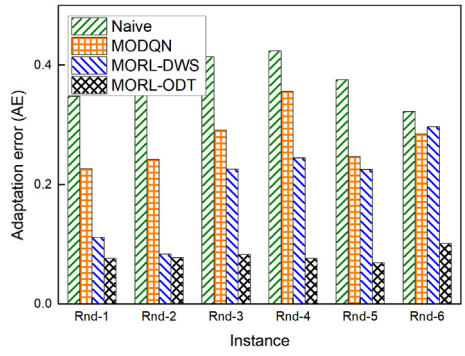

Fig. 10. Adaptation error of four algorithms.

- MORL-ODT: The proposed MORL-DWS with the TS scheme in this paper.

Fig. 8 shows the cumulative regret obtained by the four MORLs. MORL-ODT performs better than the others in all test instances as its cumulative regret in each episode is the smallest. On the contrary, Naive is the worst algorithm due to its largest cumulative regret. Naive learns the optimal Q-values for each objective and selects actions using a synthetic function. As these Q-values are updated for each objective individually, the resulting Q-value vectors are not good at capturing a promising tradeoff between objectives. It means that Naive cannot simultaneously minimize multiple objectives. MODQN is the second worst MORL with all test instances considered. Its Q-network is only trained on the current preference, and it does not take the preference as input. Thus, MODQN fails to memorize the learned policies for preferences and cannot adapt to MEC scenarios with dynamic preferences.

Unlike MODQN, MORL-DWS takes the current preference as input to the Q-network to handle the dynamics of preferences. In addition, to avoid overfitting to the current preference, MORL-DWS trains its Q-network on two preferences, i.e., the current and encountered preferences. Thanks to the two advantages above, MORL-DWS is much better than MODQN in terms of the cumulative regret. This also justifies why MORL-DWS is chosen for solving the ODT problem.

Table 9
Results of AACT (Sec.).

Instance	IMOEA/D	IDQN	Naive	MODQN	MORL-DWS	MORL-ODT
Rnd-1	5.9627	6.8004	6.8156	6.7610	6.0667	**5.6845**
Rnd-2	5.2822	5.7968	5.8158	5.7152	5.4154	**5.1779**
Rnd-3	10.9821	12.1283	12.0930	12.1308	12.0760	**10.5974**
Rnd-4	9.7879	11.4472	11.5313	11.5521	11.3988	**9.5662**
Rnd-5	15.8852	18.4217	18.4842	18.6089	18.9595	**15.4861**
Rnd-6	15.3373	16.2573	16.1351	16.9388	16.2192	**13.6783**

Meanwhile, MORL-ODT outperforms MORL-DWS in all test instances because of the TS scheme. This scheme does not treat preferences equally, i.e., more significant preferences are selected with higher probabilities. This is helpful for the Q-network to maintain the learned policies, thus improving the multi-objective optimization performance under dynamic preferences. The average episodic regret of four MORL algorithms is illustrated in Fig. 9. One can see clearly that MORL-ODT is the best among all algorithms.

Fig. 10 shows the adaptation error obtained by the four MORLs. MORL-ODT is the best among all MORLs because it always results in the lowest AE in each test instance, which means MORL-ODT better adapts to MEC scenarios with dynamic preferences than the other three MORLs. In addition, Student's t-test [44] is conducted to compare the proposed MORL-ODT with the other three MORLs in Table 8, where AER and AE values obtained in 10 runs are utilized to reflect the performance of each MORL. A two-tailed t-test with 18 degrees of freedom at a 0.05 level of significance is used in this paper. The result of comparison between Algorithm-A and Algorithm-B is shown as "+" when the former is significantly better than the latter. One can see clearly that MORL-ODT is the best among all MORLs.

We also add two state-of-the-art algorithms for performance comparison in terms of AACT, AEC, AUC, ACOI, and NU, including a metaheuristic, i.e., IMOEA/D and an SORL, i.e., IDQN, as listed below.

- IMOEA/D [26]: The improved multi-objective evolutionary algorithm based on decomposition proposed to solve the multi-objective computation offloading problem, where the application completion time and energy consumption of MDs are minimized, simultaneously.
- IDQN [38]: The improved deep Q-network based on the long short-term memory and candidate network proposed to address the offloading problem of multiple service nodes and dependent tasks in a heterogeneous MEC environment. IDQN aims at minimizing the application completion time and energy consumption of MDs, simultaneously.

Tables 9–12 show the AACT, AEC, AUC and ACOI values of all preferences in Ω. Note that the best results are in bold. In Table 9, one can observe that MORL-ODT is the best among all algorithms as it always achieves the smallest AACT. This also reflects that

Table 10
Results of AEC (J).

Instance	IMOEA/D	IDQN	Naive	MODQN	MORL-DWS	MORL-ODT
Rnd-1	4.9506	**4.8169**	4.8538	4.9270	4.9106	4.9089
Rnd-2	4.9453	4.8000	4.8216	4.9509	4.8940	**4.5892**
Rnd-3	9.9779	9.5787	9.7800	10.0221	9.5111	**9.4959**
Rnd-4	10.1407	9.8669	9.7828	10.0130	9.7588	**9.7283**
Rnd-5	14.5129	**13.7810**	14.1702	14.7254	14.8112	14.7884
Rnd-6	15.1731	15.0880	15.3230	15.4100	15.5008	**14.7008**

Table 11
Results of AUC ($).

Instance	IMOEA/D	IDQN	Naive	MODQN	MORL-DWS	MORL-ODT
Rnd-1	0.0070	0.0068	0.0065	0.0067	0.0059	**0.0056**
Rnd-2	0.0053	0.0044	0.0043	0.0041	**0.0036**	**0.0036**
Rnd-3	**0.0138**	0.0151	0.0152	0.0148	0.0150	0.0142
Rnd-4	0.0111	0.0116	0.0112	0.0110	**0.0107**	0.0108
Rnd-5	0.0225	0.0253	0.0245	0.0241	0.0241	**0.0217**
Rnd-6	**0.0230**	0.0245	0.0240	0.0245	0.0236	0.0237

Table 12
Results of ACOI.

Instance	IMOEA/D	IDQN	Naive	MODQN	MORL-DWS	MORL-ODT
Rnd-1	3.1267	3.4420	3.4533	3.4095	3.1296	**3.0995**
Rnd-2	2.9026	3.1079	3.0765	2.9867	2.8863	**2.8205**
Rnd-3	5.7579	6.2715	6.2372	6.2799	6.1518	**5.6933**
Rnd-4	5.3557	6.0516	5.9953	6.0412	5.8639	**5.2738**
Rnd-5	8.5041	9.5617	9.5726	9.6151	9.7332	**8.4234**
Rnd-6	8.0918	8.8285	8.6235	8.9099	8.6229	**7.9015**

MORL-ODT can achieve lower network latency than the other algorithms for comparison. As for the AEC collected in Table 10, MORL-ODT performs better than the other five algorithms in most test instances (except for Rnd-1 and Rnd-5). On the other hand, IDQN obtains the smallest AEC in Rnd-1 and Rnd-5. But it all leads to poor performance in terms of AACT and AUC. For example, although IDQN can achieve the smallest AEC values in Rnd-5, its AUC is the largest. If considering the AUC shown in Table 11, MORL-ODT is the best in Rnd-1, Rnd-2, and Rnd-5. IMOEA/D obtains the best AUC in Rnd-3 and Rnd-6, but it leads to poor performance with respect to AACT and AEC. For instance, although IMOEA/D achieves the smallest AUC values in Rnd-3 and Rnd-6, it results in larger AACT and AEC values than MORL-ODT. In summary, IMOEA/D, IDQN, Naive, MODQN, and MORL-DWS cannot obtain an excellent tradeoff between the three objectives, thus cannot optimize them simultaneously. While MORL-ODT cannot obtain the best performance in all test instances in terms of AEC and AUC, it is better than the other five algorithms regarding ACOI. MORL-ODT can strike a balance between $T(G)$, $E(G)$, and $C(G)$.

Table 13 shows the NU values obtained by six algorithms. One can observe that MORL-ODT is better than the other five algorithms in 4 instances as it obtains the smallest NU values. MORL-ODT is good at scheduling tasks between the MD and edge servers, in favor of network resource utilization. Thus, MORL-ODT has lower NU values than other algorithms.

In addition, Friedman test [45] is utilized for algorithm ranking. Based on the AACT, AEC, AUC, ACOI, and NU values, the average rankings of the six algorithms in all test instances are shown in Table 14. One can see that MORL-ODT achieves the best overall performance.

7. Conclusion

In this paper, we model a new ODT problem in the MEC system, where three objectives, i.e., the application completion time, energy consumption of the MD, and usage charge for edge computing, are minimized at the same time. To address the problem, we model an MOMDP with a vector-valued reward, where each component of the reward corresponds to an objective. Besides, we propose an improved MORL-DWS with the TS scheme to solve the ODT problem, namely MORL-ODT. The TS scheme makes sure that more important preferences are more likely to be selected for Q-network training, which is helpful for effectively maintaining the previously learned policies. We conduct extensive simulation experiments on six test instances with different task topologies and profiles. The simulation results demonstrate that the proposed MORL-ODT can simultaneously minimize the

Table 13
Results of NU (KB).

Instance	IMOEA/D	IDQN	Naive	MODQN	MORL-DWS	MORL-ODT
Rnd-1	3657.5849	3725.4127	3337.4554	3634.3517	3453.8618	**3257.5849**
Rnd-2	3115.1060	3821.6485	3499.3138	3226.6965	**2945.5566**	3115.1060
Rnd-3	4435.1491	4207.6328	4027.8681	4146.3094	4154.0323	**4005.1491**
Rnd-4	4076.2584	4180.1180	4100.9471	**4000.4771**	4011.9790	4076.2584
Rnd-5	4392.1684	4294.5769	4201.1184	4350.7771	4325.4000	**4192.1684**
Rnd-6	4181.5532	4770.7939	4703.3653	4334.8630	4154.9018	**4081.5532**

Table 14
Ranking of all algorithms.

Algorithm	AACT		AEC		AUC		ACOI		NU	
	Average rank	Position	Average rank	Position	Average rank	Position	Average rank	Position	Average rank	Position
IMOEA/D	2.0000	2	4.6667	5	3.3333	3	2.1667	2	4.1667	4
IDQN	4.5000	4	2.1667	2	5.3333	6	5.0000	5	5.3333	5
Naive	4.6667	5	3.0000	3	4.5000	5	4.5000	4	3.5000	3
MODQN	5.1667	6	5.1667	6	3.5000	4	5.0000	5	3.5000	3
MORL-DWS	3.6667	3	4.0000	4	2.1667	2	3.3333	3	2.6667	2
MORL-ODT	1.0000	1	2.0000	1	1.6667	1	1.0000	1	1.5000	1

three objectives above and performs better than three MORLs, one metaheuristic, and one SORL in terms of the regret, adaptation error, application completion time, energy consumption, usage charge, and network usage.

CRediT authorship contribution statement

Fuhong Song: Methodology, Conceptualization, Writing – original draft. **Huanlai Xing:** Methodology, Supervision, Writing – review & editing. **Xinhan Wang:** Methodology, Simulation. **Shouxi Luo:** Simulation, Validation. **Penglin Dai:** Investigation, Validation. **Ke Li:** Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (No. 61802319, No. 62002300), China Postdoctoral Science Foundation (No. 2019M660245, No. 2019M663552, No. 2020T130547), the Fundamental Research Funds for the Central Universities, and China Scholarship Council, P.R. China.

References

[1] E. Novak, Z. Tang, Q. Li, Ultrasound proximity networking on smart mobile devices for IoT applications, IEEE Internet Things J. 6 (1) (2019) 399–409.
[2] M. Goudarzi, H. Wu, M. Palaniswami, R. Buyya, An application placement technique for concurrent IoT applications in edge and fog computing environments, IEEE Trans. Mob. Comput. 20 (4) (2020) 1298–1311.
[3] Z. Xia, J.A. Qahouq, State-of-charge balancing of lithium-ion batteries with state-of-health awareness capability, IEEE Trans. Ind. Appl. 57 (1) (2021) 673–684.
[4] P. Mach, Z. Becvar, Mobile edge computing: a survey on architecture and computation offloading, IEEE Commun. Surv. Tutor. 19 (3) (2017) 1628–1656.
[5] A. Ndikumana, N.H. Tran, T.M. Ho, Z. Han, W. Saad, D. Niyato, C.S. Hong, Joint communication, computation, caching, and control in big data multi-access edge computing, IEEE Trans. Mob. Comput. 19 (6) (2020) 1359–1374.
[6] Y. Mao, C. You, J. Zhang, K. Huang, K. Letaief, A survey on mobile edge computing: the communication perspective, IEEE Commun. Surv. Tutor. 19 (4) (2017) 2322–2358.
[7] A. Shakarami, A. Shahidinejad, M.G. Arani, An autonomous computation offloading strategy in mobile edge computing: a deep learning-based hybrid approach, J. Netw. Comput. Appl. 178 (2021) 102974.
[8] S. Sundar, B. Liang, Offloading dependent tasks with communication delay and deadline constraint, in: Proceeding of IEEE Conference on Computer Communications, 2018, pp. 16–19.
[9] G. Zhao, H. Xu, Y. Zhao, C. Qiao, L. Huang, Offloading dependent tasks in mobile edge computing with service caching, in: Proceeding of IEEE Conference on Computer Communications, 2020, pp. 6–9.
[10] H. Peng, W. Wen, M. Tseng, L. Li, Joint optimization method for task scheduling time and energy consumption in mobile cloud computing environment, Appl. Soft Comput. 80 (2019) 534–545.
[11] A.A. Habob, O.A. Dobre, A.G. Armada, S. Muhaidat, Task scheduling for mobile edge computing using genetic algorithm and conflict graphs, IEEE Trans. Veh. Technol. 69 (8) (2019) 8805–8819.
[12] J. Chen, H. Xing, Z. Xiao, L. Xu, T. Tao, A DRL agent for jointly optimizing computation offloading and resource allocation in MEC, IEEE Internet Things J. (2021) http://dx.doi.org/10.1109/JIOT.2021.3081694.
[13] M.G. Arani, A. Souri, A.A. Rahmanian, Resource management approaches in fog computing: a comprehensive review, J. Grid Comput. 18 (2020) 1–42.
[14] A. Shakarami, M.G. Arani, A. Shahidinejad, A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective, Comput. Netw. 128 (2020) 107496.
[15] J. Yan, S. Bi, Y. Jun, A. Zhan, Offloading and resource allocation with general task graph in mobile edge computing: A deep reinforcement learning approach, IEEE Trans. Wirel. Commun. 18 (8) (2020) 5404–5419.
[16] J. Wang, J. Hu, G. Min, W. Zhan, Q. Ni, N. Georgalas, Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning, IEEE Commun. Mag. 57 (5) (2019) 64–69.
[17] C. Liu, X. Xu, D. Hu, Multiobjective reinforcement learning: A comprehensive overview, IEEE Trans. Syst. Man Cybern. Syst. 386 (3) (2015) 385–398.
[18] T.T. Nguyen, N.D. Nguyen, P. Vamplew, S. Nahavandi, R. Dazeley, C.P. Lim, A multi-objective deep reinforcement learning framework, Eng. Appl. Artif. Intell. 96 (2020) 1–12.
[19] A. Abels, D.M. Roijers, T. Lenaerts, A. Now´e, D. Steckelmacher, Dynamic weights in multi-objective deep reinforcement learning, in: Proceedings of ACM International Conference on Machine Learning, 2019, pp. 1–10.
[20] D.M. Roijers, S. Whiteson, Multi-Objective Decision Making, Morgan & Laypool, San Rafael, CA, USA, 2017.
[21] W. Zhang, Y. Wen, Energy-efficient task execution for application as a general topology in mobile cloud computing, IEEE Trans. Cloud Comput. 6 (3) (2018) 708–718.
[22] M. Maray, A. Jhumka, A. Chester, M. Younis, Scheduling dependent tasks in edge networks, in: Proceedings of IEEE International Performance Computing and Communications Conference, 2019, pp. 1–4.
[23] S.E. Mahmoodi, R.N. Uma, K.P. Subbalakshm, Optimal joint scheduling and cloud offloading for mobile applications, IEEE Trans. Cloud Comput. 7 (2) (2019) 301–313.
[24] Y. Sahni, J. Cao, L. Yang, Y. Ji, Multihop offloading of multiple DAG tasks in collaborative edge computing, IEEE Internet Things J. 8 (6) (2021) 4893–4905.
[25] B. Huang, Z. Li, P. Tang, S. Wang, J. Zhao, H. Hu, W. Li, V. Chang, Security modeling and efficient computation offloading for service workflow in mobile edge computing, Future Gener. Comput. Syst. 97 (2019) 755–774.
[26] F. Song, H. Xing, S. Luo, D. Zhan, P. Dai, R. Qu, A multiobjective computation offloading algorithm for mobile-edge computing, IEEE Internet Things J. 7 (9) (2020) 8780–8799.
[27] Y. Xie, Y. Zhu, Y. Wang, Y. Cheng, R. Xu, A.S. Sani, D. Yuan, Y. Yang, A novel directional and non-local-convergent particle swarm optimization based workflow scheduling in cloud–edge environment, Future Gener. Comput. Syst. 97 (2019) 361–378.
[28] K.K.V. Mnih, D. Silver, Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529–533.
[29] J. Wang, J. Hu, G. Min, A.Y. Zomaya, N. Georgalas, Fast adaptive task offloading in edge computing based on meta reinforcement learning, IEEE Trans. Parallel Distrib. Syst. 32 (1) (2021) 242–253.
[30] Q. Wu, Z. Wu, Y. Zhuang, Y. Cheng, Adaptive tasks scheduling with deep reinforcement learning, in: Proceedings of IEEE International Conference on Algorithms and Architectures for Parallel Processing, 2018, pp. 477–490.
[31] A. Zhu, S. Guo, M. Ma, H. Feng, B. Liu, X. Su, M. Guo, Q. Jiang, Computation offloading for workflow in mobile edge computing based on deep Q-learning, in: Proceedings of IEEE Wireless and Optical Communications Conference, 2019, pp. 1–5.
[32] Z. Tang, J. Lou, F. Zhang, W. Jia, Dependent task offloading for multiple jobs in edge computing, in: Proceedings of IEEE International Conference on Computer Communications and Networks, 2020, pp. 1–9.
[33] B. Wang, H. Li, Z. Lin, Y. Xia, Temporal fusion pointer network-based reinforcement learning algorithm for multi-objective workflow scheduling in the cloud, in: Proceedings of IEEE International Joint Conference on Neural Networks, 2020, pp. 1–8.
[34] A. Shahidinejad, M.G. Arani, Joint computation offloading and resource provisioning for edge-cloud computing environment: A machine learning-based approach, Softw. - Pract. Exp. 50 (2020) 2212–2230.
[35] F. Jazayeri, A. Shahidinejad, M.G. Arani, Autonomous computation offloading and auto-scaling the in the mobile fog computing: A deep reinforcement learning-based approach, J. Ambient Intell. Humaniz. Comput. 12 (2021) 8265–8284.
[36] F. Jazayeri, A. Shahidinejad, M.G. Arani, A latency-aware and energy-efficient computation offloading in mobile fog computing: A hidden Markov model-based approach, J. Supercomput. 77 (2021) 4887–4916.
[37] G. Qu, H. Wu, R. Li, P. Jiao, DMRO: a deep meta reinforcement learning-based task offloading framework for edge-cloud computing, IEEE Trans. Netw. Serv. Manag. (2021) http://dx.doi.org/10.1109/TNSM.2021.3087258.
[38] H. Lu, C. Gu, F. Luo, W. Ding, X. Liu, Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning, Future Gener. Comput. Syst. 102 (2020) 847–861.
[39] P. Nguyen, L.B. Le, Joint computation offloading, SFC placement, and resource allocation for multi-site MEC systems, in: Proceedings of IEEE Wireless Communications and Networking Conference, 2020, pp. 1–6.
[40] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, M. Bennis, Performance optimization in mobile-edge computing via deep reinforcement learning, in: Proceedings of IEEE Vehicular Technology Conference, 2018, pp. 1–6.
[41] F. Suter, DAG generation program, 2010, http://www.loria.fr/~suter/dags.html.
[42] Q. Zhang, H. Li, MOEA/D: A multiobjective evolutionary algorithm based on decomposition, IEEE Trans. Evol. Comput. 11 (6) (2007) 712–731.

[43] H. Mossalam, Y.M. Assael, D.M. Roijers, S. Whiteson, Multi-objective deep reinforcement learning, 2016, CoRR, abs/1610.02707, http://arxiv.org/abs/1610.02707.
[44] R.E. Walpole, R.H. Myers, S.L. Myers, K. Ye, Probability and Statistics for Engineers and Scientists, Pearson Educ., New Jersey, NJ, USA, 2007.
[45] M. Friedman, A comparison of alternative tests of significance for the problem of m rankings, Ann. Math. Stat. 11 (1) (1940) 86–92.

Fuhong Song, received his M. Eng. degree in computer technology from Southwest Jiaotong University, Chengdu, China, in 2018. He is currently pursuing the Ph.D. degree in computer science and technology at Southwest Jiaotong University. His research interests include edge computing, multi-objective optimization and reinforcement learning.

Huanlai Xing, received his B. Eng. degree in communications engineering from Southwest Jiaotong University, Chengdu, China, in 2006; his M.Sc. degree in electromagnetic fields and wavelength technology from Beijing University of Posts and Telecommunications, Beijing, China, in 2009; and his Ph.D. degree in computer science from University of Nottingham, Nottingham, U.K., in 2013. He is an Associate Professor with the School of Computing and Artificial Intelligence, Southwest Jiaotong University. His research interests include evolutionary computation, cloud computing, multi-objective optimization, network function virtualization and software defined networks. He has authored and co-authored over 30 peer-reviewed journal and conference papers. Dr. Xing is a Member of IEEE and ACM.

Xinhan Wang, received his B. Eng. degree in Computer Science and Technology from Southwest University, Chongqing, China, in 2016. He is currently pursuing the Ph.D. degree in the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu, China. His research interests include evolutionary computation, software defined networks and network function virtualization.

Shouxi Luo, received the bachelor's degree in communication engineering and the Ph.D. degree in communication and information systems from the University of Electronic Science and Technology of China, in 2011 and 2016, respectively. He is currently a Lecturer with the School of Computing and Artificial Intelligence, Southwest Jiaotong University. His research interests include data center networks, software-defined networking, and networked systems.

Penglin Dai, received the B.S. degree in mathematics and applied mathematics and the Ph.D. degree in computer science from Chongqing University, Chongqing, China, in 2012 and 2017, respectively. He is currently an Assistant Professor with the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu, China. His research interests include intelligent transportation systems and vehicular cyber–physical systems.

Ke Li, received her Ph.D. degree in communication and information systems from the University of Electronic Science and Technology of China, in 2012. She is currently a Lecturer with the School of Computing and Artificial Intelligence, Southwest Jiaotong University. Her research interests include intelligent transportation systems and vehicular networks.